

Design of Dependable Computing Systems

Design of Dependable Computing Systems

by

Jean-Claude Geffroy

Gilles Motet

*Institut National des Sciences Appliquées,
Toulouse, France*



SPRINGER-SCIENCE+BUSINESS MEDIA, B.V.

A C.I.P. Catalogue record for this book is available from the Library of Congress.

ISBN 978-90-481-5941-3 ISBN 978-94-015-9884-2 (eBook)
DOI 10.1007/978-94-015-9884-2

Translated from the Russian language by Irene Aleksanova.

Revised and translated version of Linear Programming by F.P. Vasilyev and A. Yu. Ivanitskiy,
published in the Russian language by Factorial, Moscow, 1998

Printed on acid-free paper

All Rights Reserved

© 2002 Springer Science+Business Media Dordrecht

Originally published by Kluwer Academic Publishers in 2002

Softcover reprint of the hardcover 1st edition 2002

No part of the material protected by this copyright notice may be reproduced or
utilized in any form or by any means, electronic or mechanical,
including photocopying, recording or by any information storage and
retrieval system, without written permission from the copyright owner.

Contents

Preface	xv
CHAPTER 1. INTRODUCTORY ELEMENTS: DEPENDABILITY ISSUES	1
1.1 Quality	1
1.1.1 Quality Needs of Computer Systems	1
1.1.2 Quality Attributes	2
1.2 Dependability	3
1.2.1 Product Failures and their Consequences	3
1.2.2 Failure Causes	4
1.2.3 Taking Faults into Account	7
1.2.4 Definitions of Dependability	9
1.3 Means of Dependability	10
1.3.1 Evolution	10
1.3.2 Means	13
1.4 Summary	13
FIRST PART. DESTRUCTIVE MECHANISMS	15
CHAPTER 2. GENERAL CONTEXT	17
2.1 Application Context	17
2.2 Life Cycle	21
2.2.1 Principles	21
2.2.2 Specification	22
2.2.3 Design	24
2.2.4 Production	28
2.2.5 Operation	29

2.3	Product Model	29
2.3.1	Product Structure and Functioning	30
2.3.2	Hierarchy	31
2.3.3	Examples	32
2.3.4	Refinement Process and Primitive Components	33
2.4	Logical Part of a Drinks Distributor	34
2.4.1	Specifications	35
2.4.2	Design	36
2.4.3	Production	38
2.4.4	Operation	38
CHAPTER 3. FAILURES AND FAULTS		39
3.1	Failures	39
3.1.1	Definition	39
3.1.2	Characterization of Failures	42
3.2	Faults	44
3.2.1	Difficulties in Identifying the Causes of a Failure	44
3.2.2	Fault Characterization	45
3.2.3	Fault Origin	46
3.2.4	Nature of the Fault	48
3.3	Faults Occurring in the Life Cycle	51
3.3.1	Specification and Design Faults	52
3.3.2	Production Faults	56
3.3.3	Operational Faults	58
3.4	Examples of Functional Faults Altering a Drinks Distributor	60
3.4.1	Description of the Product	60
3.4.2	Faults Due to Functional Specifications	61
3.4.3	Faults Due to Technological Constraints	61
3.4.4	Design Faults	62
3.5	Interests and Limits of Fault Classes	63
3.5.1	Simplified Classification	63
3.5.2	Limitations of the Classification	65
3.5.3	Protection Against Faults and their Effects	65
3.6	Exercises	66
CHAPTER 4. FAULTS AND THEIR EFFECTS		69
4.1	Internal Effects	69
4.1.1	Fault	69
4.1.2	Error	71
4.1.3	Error Propagation	73
4.1.4	Latency	75
4.2	External Effects: Consequences	77
4.2.1	External Consequences of Faults	77
4.2.2	Inertia of the Functional Environment	80

4.2.3	Completeness and Compatibility	80
4.2.4	Influence of the Functional Environment: Emergence	82
4.3	Conclusion on the Effects of Faults	83
4.4	Exercises	85
CHAPTER 5.	FAULT AND ERROR MODELS	89
5.1	Definitions	89
5.1.1	Structural and Behavioral Properties	89
5.1.2	Structural Properties	90
5.1.3	Behavioral Properties	91
5.2	Significant Fault and Error Models	92
5.2.1	Faults and Errors at Different Representation Levels	92
5.2.2	Hardware Fault/Error Models	94
5.2.3	Software Fault and Error Models	101
5.3	Fault and Error Model Assessment	105
5.3.1	Assessment Criteria	105
5.3.2	Relations Between Fault/Error Models and Failures	107
5.4	Analysis of Two Simple Examples	109
5.4.1	First example: an Hardware Full Adder	109
5.4.2	Second Example: a Software Average Function	111
5.5	Exercises	115
SECOND PART. PROTECTIVE MECHANISMS		119
CHAPTER 6.	TOWARDS THE MASTERING OF FAULTS AND THEIR EFFECTS	121
6.1	Three Approaches	121
6.2	Fault Prevention	123
6.2.1	During the Specification	123
6.2.2	During the Design	124
6.2.3	During the Production	124
6.2.4	During the Operation	125
6.3	Fault Removal	127
6.3.1	General Notions	127
6.3.2	During Specification and Design	129
6.3.3	During the Production	133
6.3.4	During the Operation	134
6.4	Fault Tolerance	135
6.4.1	Failure Prevention by Masking	136
6.4.2	Error Detection and Correction	136
6.4.3	Fail-Safe Techniques	137
6.4.4	Resulting Fault Tolerance Classes	138
6.5	Dependability Means and Assessment	138
6.6	Conclusion	140

CHAPTER 7. DEPENDABILITY ASSESSMENT	141
7.1 Quantitative and Qualitative Assessment	141
7.1.1 Quantitative Assessment	141
7.1.2 Qualitative Assessment	143
7.1.3 Synthesis	143
7.2 Reliability	145
7.2.1 General Characteristics of the Reliability of Electronic Systems	145
7.2.2 Reliability Models	146
7.2.3 Failure Rate Estimation	148
7.2.4 Reliability Evolution	148
7.3 Testability	149
7.4 Maintainability	150
7.4.1 Maintenance	150
7.4.2 Maintainability	152
7.4.3 Reliability and Maintainability	153
7.5 Availability	154
7.6 Safety	155
7.7 Security	157
7.8 Synthesis of the Main Criteria	157
7.9 Quantitative Analysis Tools at System Level	159
7.9.1 Fault Simulation	159
7.9.2 Reliability Block Diagrams	160
7.9.3 Non-Deterministic State Graph Models	162
7.10 Inductive Qualitative Assessment: Failure Mode and Effect Analysis	164
7.10.1 Principles	164
7.10.2 Means	166
7.10.3 FMECA	167
7.11 Deductive Qualitative Assessment: Fault Tree Method	168
7.11.1 Principles	168
7.11.2 Software Example	169
7.11.3 Use of the FTM	171
7.12 Exercises	171
CHAPTER 8. REDUNDANCY	175
8.1 Functional and Structural Redundancy	176
8.1.1 Linguistic Redundancy	176
8.1.2 Redundancy of Computer Systems	177
8.2 Functional Redundancy	179
8.2.1 Static Functional Domains	180
8.2.2 Dynamic Functional Domains	182
8.2.3 Generalization of Functional Redundancy	185

8.2.4	Redundancy and Module Composition	186
8.3	Structural Redundancy	187
8.3.1	Definition and Illustration	187
8.3.2	Active and Passive Redundancy	188
8.3.3	Separable Redundancy	193
8.3.4	Summary of the Various Redundancy Forms	195
8.4	Exercises	195
THIRD PART. FAULT AVOIDANCE MEANS		199
CHAPTER 9. AVOIDANCE OF FUNCTIONAL FAULTS DURING SPECIFICATION		201
9.1	Introduction	201
9.1.1	Specification Phase	201
9.1.2	Validation and Verification	202
9.2	Fault Prevention During the Requirement Expression	204
9.2.1	Introduction	204
9.2.2	Help in the Capturing of Needs	204
9.2.3	Expression Aid	205
9.2.4	Evaluation of a Method	207
9.3	Fault Avoidance During the Specification Phase	209
9.3.1	Fault Prevention: Valid Method	209
9.3.2	Fault Removal: Verification of the Specifications	211
9.4	Review Techniques	214
9.4.1	Principles	214
9.4.2	Walkthrough	215
9.4.3	Inspection	215
9.5	Exercise	217
CHAPTER 10. AVOIDANCE OF FUNCTIONAL FAULTS DURING DESIGN		219
10.1	Principles	219
10.2	Prevention by Design Model Choice	222
10.3	Prevention by Design Process Choice	223
10.3.1	General Considerations	223
10.3.2	Design Guide	224
10.3.3	Expression Guide	225
10.4	Fault Removal	229
10.4.1	Verification with the Specifications	229
10.4.2	Fault Removal without Specifications	238
10.5	Functional Test	240
10.5.1	Input Sequence	240
10.5.2	Output Sequence	243
10.5.3	Functional Diagnosis	245
10.5.4	Analysis of an Arithmetic Unit	247

10.6	Formal Proof Methods	248
10.6.1	Inductive Approach and Symbolic Execution	248
10.6.2	Deductive Approach and FTM	251
10.7	Exercises	253
CHAPTER 11.	PREVENTION OF TECHNOLOGICAL FAULTS	257
11.1	Parameters of the Prevention of Technological Faults	257
11.1.1	Hardware Technology	258
11.1.2	Software Technology	258
11.1.3	Prevention of Technological Faults	260
11.2	Action on the Product	261
11.2.1	Hardware Technology	261
11.2.2	Software Technology	265
11.3	Action on the Environment	272
11.3.1	Hardware Technology	272
11.3.2	Software Technology	273
11.4	Exercises	276
CHAPTER 12.	REMOVAL OF TECHNOLOGICAL FAULTS	279
12.1	Off-Line Testing	279
12.1.1	Context of Off-Line Testing	280
12.1.2	Different Kinds of Tests and Testers	281
12.2	Logical Testing	288
12.2.1	Logical Testers	288
12.2.2	Test Parameters	291
12.2.3	Production Testing	292
12.2.4	Maintenance Testing	296
12.3	Principles of Logical Test Generation	302
12.3.1	Logical Testing	302
12.3.2	Determination of Input Vectors Testing a Fault	307
12.3.3	Fault Grading	307
12.3.4	Test Pattern Generation of Combinational Systems	314
12.3.5	Test of Sequential Systems	316
12.4	Exercises	320
CHAPTER 13.	STRUCTURAL TESTING METHODS	323
13.1	Generation of Logical Test by a Gate Level Structural Approach	323
13.2	Test Generation for a Given Error	325
13.2.1	Principles of the Method	325
13.2.2	Activation and Backward Propagation	326
13.2.3	Forward Propagation	327
13.2.4	Justification	329
13.2.5	Complete Study of a Small Circuit	329

13.2.6	Test of Structured Circuits	332
13.3	Determination of the Faults/Errors Detected by a Given Test Vector	333
13.3.1	Principles of the Method	333
13.3.2	Study of a Small Circuit	335
13.4	Diagnosis of a Test Sequence	336
13.4.1	General Problem of the Diagnosis	336
13.4.2	Study of a Small Circuit	337
13.5	Influence of Passive Redundancy on Detection and Diagnosis	339
13.6	Detection Test without Error Model. Application to Software	340
13.6.1	The Problem of Structural Test without Error Model	340
13.6.2	Statement Test	342
13.6.3	Branch & Path Test	343
13.6.4	Condition & Decision Test	345
13.6.5	Finite State Machine Identification	346
13.7	Diagnosis without Fault Models	346
13.7.1	Principles	346
13.7.2	Highlight the Erroneous Situations	347
13.7.3	Elaborate the Hypotheses	349
13.7.4	Confirm the Hypotheses	350
13.7.5	Verify the Hypotheses	350
13.8	Mutation Test Methods	351
13.8.1	Principles and Pertinence of Mutation Methods	351
13.8.2	Mutation Testing Technique	352
13.9	Exercises	354
CHAPTER 14. DESIGN FOR TESTABILITY		361
14.1	Introduction	361
14.1.1	Test Complexity	361
14.1.2	General Principles of Design For Testability	362
14.2	Ad Hoc Approach to DFT	367
14.2.1	Guidelines	367
14.2.2	Instrumentation: Data Recording	373
14.2.3	Exception Mechanisms: Error Propagation	374
14.3	Design of Systems Having Short Test Sequences	377
14.3.1	Illustration on Electronic Products	377
14.3.2	Illustration on Software Applications	379
14.4	Built-In Test (BIT)	380
14.4.1	Introduction	380
14.4.2	The FIT PLA	380
14.4.3	Scan Design and LSSD	383
14.4.4	Boundary Scan	385
14.4.5	Discussion about BIT Evolution	387

14.5	Built-In Self-Test (BIST)	388
14.5.1	Principles	388
14.5.2	Test Sequence Generation and Signature Analysis	389
14.6	Towards On-Line Testing	392
14.6.1	To Place the Tester in the Application Site	392
14.6.2	<i>In-situ</i> Maintenance Operation	392
14.6.3	Integration of the Tester to the Product's Activity	393
14.7	Exercises	393
FOURTH PART. FAULT TOLERANCE MEANS		397
CHAPTER 15. ERROR DETECTING AND CORRECTING CODES		399
15.1	General Context	399
15.1.1	Error Model	399
15.1.2	Redundant Coding	402
15.1.3	Application to Error Detection and Correction	403
15.1.4	Limitations of our Study	404
15.2	Definitions	405
15.2.1	Separable and Non-Separable Codes	405
15.2.2	Hamming Distance	406
15.2.3	Redundancy and Efficiency	408
15.3	Parity Check Codes	409
15.3.1	Single Parity Code	409
15.3.2	Multiple Parity Codes	409
15.4	Unidirectional Codes	416
15.4.1	M-out-of-n Codes	417
15.4.2	Two-Rail Codes	418
15.4.3	Berger Codes	418
15.5	Arithmetic Codes	419
15.5.1	Limitations of the Hamming Distance	419
15.5.2	Residual Codes	420
15.6	Application of EDC Codes to Different Classes of Systems	422
15.7	Exercises	423
CHAPTER 16. ON-LINE TESTING		427
16.1	Two Approaches of On-Line Testing	427
16.2	Discontinuous Testing	428
16.2.1	External Tester	428
16.2.2	Test Performed by One of the Regulators	430
16.2.3	Test Distributed Between the Regulators	430
16.2.4	Precautions	432
16.3	Continuous Testing: Self-Testing	433
16.3.1	Principles	433
16.3.2	Use of Functional Redundancy	436

16.3.3	Use of Structural Redundancy	441
16.4	Exercises	447
CHAPTER 17.	FAIL-SAFE SYSTEMS	451
17.1	Risk and Safety	452
17.1.1	Seriousness Classes	452
17.1.2	Risk and Safety Classes	453
17.1.3	Fail-Safe Systems	456
17.2	Fail-Safe Techniques	457
17.2.1	Intrinsic Safety	457
17.2.2	Safety by Structural Redundancy	459
17.2.3	Self-Testing Systems and Fail-Safe Systems	465
17.2.4	Fail-Safe Applications	466
17.3	Exercises	467
CHAPTER 18.	FAULT-TOLERANT SYSTEMS	469
18.1	Introduction	469
18.1.1	Aims	469
18.1.2	From Error Detection Towards Fault Tolerance	470
18.2	N-Versions	472
18.2.1	Principles	472
18.2.2	Realization of the Duplicates and the Voter	473
18.2.3	Performance Analysis	475
18.3	Backward Recovery	476
18.3.1	Principles and Use	476
18.3.2	Recovery Cache	478
18.3.3	Recovery Points	479
18.4	Forward Recovery	482
18.4.1	Principles	482
18.4.2	Recovery Blocks	482
18.4.3	Termination Mode	483
18.5	Comparison	485
18.5.1	Similarities	485
18.5.2	Differences	487
18.5.3	Use of Multiple Techniques	490
18.6	Impact on the Design	493
18.7	Some Application Domains	496
18.7.1	Watchdog and Reset	496
18.7.2	Avionics Systems	496
18.7.3	Data Storage	498
18.7.4	Data Transmission	503
18.8	Exercises	508

CHAPTER 19. CONCLUSIONS	511
19.1 Needs and Impairments	512
19.1.1 Dependability Needs	512
19.1.2 Dependability Impairments	513
19.2 Protective Means	516
19.2.1 Fault Prevention	516
19.2.2 Fault Removal	517
19.2.3 Fault Tolerance	519
19.3 Dependability Assessment	520
19.3.1 Quantitative Approaches	520
19.3.2 Qualitative Approaches	524
19.4 Choice of Methods	525
Appendix A. Error Detecting and Correcting Codes	527
Appendix B. Reliability Block Diagrams	529
Appendix C. Testing Features of a Microprocessor	535
Appendix D. Study of a Software Product	539
Appendix E. Answer to the Exercises	543
Glossary	605
References	651
Index	657

Preface

This book analyzes the causes of failures in computing systems, their consequences, as well as the existing solutions to manage them. The domain is tackled in a progressive and educational manner with two objectives:

1. The mastering of the basics of dependability domain at system level, that is to say independently of the technology used (hardware or software) and of the domain of application.
2. The understanding of the fundamental techniques available to prevent, to remove, to tolerate, and to forecast faults in hardware and software technologies.

The first objective leads to the presentation of the general problem, the fault models and degradation mechanisms which are at the origin of the failures, and finally the methods and techniques which permit the faults to be prevented, removed or tolerated. This study concerns logical systems in general, independently of the hardware and software technologies put in place. This knowledge is indispensable for two reasons:

- A large part of a product's development is independent of the technological means (expression of requirements, specification and most of the design stage). Very often, the development team does not possess this basic knowledge; hence, the dependability requirements are considered uniquely during the technological implementation. Such an approach is expensive and inefficient. Indeed, the removal of a preliminary design fault can be very difficult (if possible) if this fault is detected during the product's final testing.

- The specific dependability techniques applied at technological level (hardware, software) are often issued from general common principles. It is useful to understand these principles in order to better understand and apply the techniques.

To achieve the second objective, we approach the main techniques associated with the two technologies involved in the creation of a computing system: the hardware and the software. The joint study of these two technologies is indispensable. As a matter of fact, what is the interest of having sophisticated methods dedicated to software if the microprocessor or the memory fails? What is the use of having sophisticated methods dedicated to hardware if the control or supervision program does not function properly? The originality of our approach resides in this dual vision of systems.

Different, complementary and sometimes even antagonistic, these two facets are both necessary to manage real industrial projects. The knowledge of their particularities is indispensable: for example, the characteristics of certain faults are specific to one of these technologies. Thus, specific techniques have been developed. In addition, it is necessary to master jointly these two technologies. For example, the technique of the replication of the hardware/software control system of the first flight of Ariane V launcher was well adapted to the tolerance of a hardware fault occurring in one module. When such event occurs, the replicate module takes control of the faulty module. Unfortunately, the presence of a fault in the software could not be tolerated with this technique, as the resumption of execution by the second module affected by the same fault provoked the same failure!

Of course, knowledge on dependability is absolutely necessary to develop *critical systems*, as their failures can have dramatic consequences. This knowledge is, according to us, necessary today for all engineers involved in computerized system development projects, whether destined for control, supervision, human - machine interaction, communication, or data processing in general. Indeed, in our technological world, failures are less and less accepted by the users, even in the case of simple applications such as a text processing. Consequently, dependability science is rapidly developing, proposing new methods, techniques and tools in both hardware and software domains. Our objective is to provide the reader with a basic knowledge of dependability notions and techniques. This will naturally be useful to students, but it may also interest specialists of specific methods, in order to place their knowledge in the general context of the means offered by the domain of dependability.

Therefore, this book addresses a large public, including undergraduate and post-graduate students, researchers, as well as technicians, engineers or

managers involved in the development or maintenance of computing systems. Providing the basics of dependability principles, models, methods and applications, this book should allow the reader to then approach successfully, in specialized books, the more technical aspects on particular means of preventing, removing and tolerating faults. Many examples and exercises (with their correction) illustrate the principles and methods presented.

Organization of the Book

This book is organized into 19 chapters structured into four parts: the *Destructive Mechanisms*, the *Protective Mechanisms*, the *Fault Avoidance Means*, and the *Fault Tolerance Means*. A fifth part, not detailed hereafter, contains several technical appendices, the detailed solutions of the exercises proposed in the chapters, a glossary of all technical keywords, and a list of bibliographical references.

First Part. Destructive Mechanisms

A good understanding of mechanisms which we qualify as *destructive*, that is to say at the origin of failures, is fundamental in order to choose the appropriate antagonistic mechanisms, which we qualify as *protective*. This first part therefore introduces the main issues of dependability and explains the basic notions and definitions of destructive mechanisms: faults, errors, failures, and external consequences. This part is organized into four chapters:

- Chap 2. General Context
- Chap 3. Failures and Faults
- Chap 4. Faults and their Effects
- Chap 5. Fault and Error Models

Second Part. Protective Mechanisms

The second part tackles protective mechanisms, that is to say mechanisms aiming at preventing failure occurrences. In a first chapter, the three approaches to *fault prevention*, *fault removal*, and *fault tolerance* are analyzed, and the principal methods and techniques are organized into several groups according to the level of redundancy they imply. Then, the dependability assessment methods are introduced. First, the quantitative approaches are presented; they aim at defining measurements of the reliance which can be placed in the services provided by the system. Several evaluation criteria of dependability are explained: *reliability*, *testability*, *maintainability*, *availability*, and *safety*. Secondly, qualitative approaches are presented; they handle specific failures to assess their cause and effects.

Redundancy plays a large role in dependability. This notion is necessary in order to master protection techniques and is therefore analyzed, introducing various functional and structural forms. This second part is organized into three chapters:

- Chap 6. Towards the Mastering of Faults and their Effects
- Chap 7. Dependability Assessment
- Chap 8. Redundancy

Third Part. Fault Avoidance Means

In this part, the principal groups of techniques and methods to prevent and remove faults are studied and compared. These means concern the specification and design steps as well as the technological implementation. Their aim is to avoid the presence of faults in the delivered product. These techniques make use of the principles defined in the second part.

This part is organized into 6 chapters:

- Chap 9. Avoidance of Functional Faults During Specification
- Chap 10. Avoidance of Functional Faults During Design
- Chap 11. Prevention of Technological Faults
- Chap 12. Removal of Technological Faults
- Chap 13. Structural Testing Methods
- Chap 14. Design For Testability

Fourth Part. Fault Tolerance Means

In this part, we consider techniques and methods useful to develop fault-tolerant systems. This subject is introduced progressively. Firstly, we examine specific techniques used to detect and to correct errors on data. Then, we present the general techniques to detect errors ‘on-line’, that is to say, at run-time. Afterwards, we study error handling techniques, to avoid catastrophic failures (Fail-Safe systems), and finally to avoid all failures (Fault-Tolerant systems).

This part is organized into 5 chapters:

- Chap 15. Error Detecting and Correcting Codes
- Chap 16. On-Line Testing
- Chap 17. Fail-Safe Systems
- Chap 18. Fault-Tolerant Systems

Finally, Chapter 19 concludes the book, providing an overview of its contents and correlating the various aspects of dependability.

Note. The keywords or fundamental expressions are marked in bold and italic characters the first time they are defined, or when they are developed.

All these keywords are then collected in the Glossary, in the fifth part of the book.

Acknowledgements

This book results from many years of teaching and research activities in hardware and software Computer Sciences, as professors at the *Institut National des Sciences Appliquées of Toulouse* (INSAT). We would like to thank the colleagues and the students of the department of Electrical Engineering and Computer Sciences of INSAT who helped us in the pedagogical aspects of the book. This work also gained from the research conducted at the LESIA laboratory with academic and industrial partners. Many examples are issued from these collaborations.

We also would like to acknowledge the Inter-Editions publisher who allowed us to integrate in this book, elements previously presented in a book we wrote in French in 1998.

FIRST PART

DESTRUCTIVE MECHANISMS

This part aims at dismantling the destructive mechanisms which hamper the correct functioning of a product in the context of its application. Frequently named *impairments*, these mechanisms involve a succession of events: faults - errors - failures and their external consequences on the application.

First of all, in Chapter 2 we consider the general context of the life cycle of an electronic manufactured product implemented by hardware and/or software technologies. Then, in Chapter 3 we observe that the behavior of a product can be altered by failures; we identify the different possible causes of these problems, called faults, inside the product or in its environment. In Chapter 4, we analyze and formalize the effects of these faults inside the product (errors) and then outside the product (external consequences). Finally, in Chapter 5, we present the principal fault and error models, focusing on hardware and software technologies.

The principles and concepts associated with destructive mechanisms rely on the definitions and standards of the international scientific community. The terms and their definitions introduced in this part come from studies unifying the basic notions from hardware and software domains. The vocabulary results mainly from the following references: the ISO 8402 standard "Quality Management and Quality Assurance - Vocabulary", and the two books "Dependability. Basic Concepts and Terminology" by J-C. Laprie *et Al* editors (Springer-Verlag, 1992) and "Safeware. System Safety and Computers" by N. Leveson (Addison-Wesley, 1995).

All the problems raised in this part will be answered in the second part of this book which examine protective mechanisms, called *dependability means*. The most significant practical techniques associated with these protective mechanisms will be detailed in the third part (fault avoidance means) and the fourth part (fault tolerance means).

Chapter 1

Introductory Elements: Dependability Issues

1.1 QUALITY

1.1.1 Quality Needs of Computer Systems

The growth of the technical and scientific knowledge in our society stimulates the growth of new manufactured products, reducing the costs and delays of design and production, and improving the global quality of our life. Moreover, the consumers who demand more and more services encourage this innovation. The sophistication of automobiles is a good example of this evolution: assisted braking and grip, reduced gas consumption by a better optimization of engine performance, road navigation and choice of optimal routes, etc. What is more, the knowledge and behavior of consumers is becoming increasingly demanding about prices, of course, but equally regarding the quality of services provided by the chosen products. This notion of quality associated with manufacturing goods is progressively becoming more refined and standardized. It is now imposed on all designers and manufacturers as an essential factor in the success of their products. Therefore, the ISO 8402 standard defines *quality* as:

The totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs.

These needs to be satisfied include new services which are offered to the users, but also a better guarantee of the correctness of existing services. In particular, the demand for quality is justified by the rapid and continuous growth of the responsibilities which man entrusts to manufactured products, and more particularly to computing systems. For example, the piloting of

aircrafts is progressively transferred from a human pilot to an assistance system, and then to an automatic piloting system. The quality of such systems, in terms of correctness guarantee, is obviously as vital to the air companies buying the aircrafts as to the passengers using them. The use of computing systems for control and supervision is overflowing the specific areas for which its use was traditionally reserved, such as the manufacturing, rail and air transportation, aerospace, nuclear and military industries. It is now progressively being used in all sectors of our society. Here we quote a few examples in order to show the wide variety of these domains.

- Agriculture and rearing: irrigation, treatment, conditioning, cattle feeding and incubating apparatuses, etc.
- Towns and cities: automatic management, control and signaling used by cash dispensers, public transportations such as buses and the subway, automobile traffic, car parks, etc.
- Communications: telephones, radio, television, local and global computing networks such as the Internet, GPS (Global Positioning System), etc.
- Medical domain: measuring and analysis equipment, ambulatory and prosthesis apparatuses, cardiac stimulators and 'on-line' following of illnesses, etc.
- Automotive electronics industry: ABS (Antilock Braking System), suspension systems, computerized ignition, fuel injection, route guiding, dash board computers, etc.
- And even in our homes: lifts, electrical household equipment, hi-fis, computer games, domestic alarm systems, etc.

1.1.2 Quality Attributes

The growth of responsibilities entrusted in such electronic products requires an increasing of their correctness, because of the consequences that can result from anomalies occurring during their functioning. The general meaning of 'quality of a manufactured product' covers many other aspects, both internal and external to the product. Widely varied features such as the price, performance, manufacturing time, weight, consumption, ergonomics, reliability, safety, are considered as criteria external to the system. They come both from the client's point of view (for example, an airline company which buys and uses an airplane) and the user's point of view (for example, the pilot or the passengers).

Other features such as modularity, readability and changeability (the facility with which a product's design can be modified) are considered as criteria internal to the product. They concern the features relative to the development of a product and not to the services it provides. If the internal quality of a product is an important need for the manufacturer, this will only indirectly affect the users. For example, a badly structured design (poor internal quality) can still result in a product which operates perfectly well. However, this could make ultimate modifications of functionality difficult, and therefore increase the costs and delays of these modifications, or even increase the risks of obtaining a failing system (poor external quality).

In this book, we are mainly interested in external quality. More precisely, our presentation focuses on functional quality, that is to say the adequacy of the function actually provided by a product with the function expected by its user. The function characterizes the behavior of a product placed in interaction with other systems (industrial processes, human operators, etc.). This notion is to be disassociated from the other non-functional features of a product. For example, the color and shape of a telephone do not affect the telephone function which is to establish communication between users.

1.2 DEPENDABILITY

1.2.1 Product Failures and their Consequences

Experience reveals that non-desired behaviors of products can occur whilst being used. These products show a temporary or permanent alteration of the function they are meant to carry out. This divergence is called *failure*. The occurrence of failures seems to be an unavoidable situation. Each one is confronted daily, sometimes harshly, with the relentless law of 'growing entropy', a law of physical sciences which seems to govern the universe. This law, which tends to disorganize what one already finds difficult to organize, has been formulated several times: the law of maximum trouble, also called the law of 'battered toast' (which always lands on the battered side down!), and alternatively known as Murphy's Law, which essentially says that when several possibilities exist it is always the worst which happens! All structured systems, such as natural biological systems and manufactured artificial products, have a tendency to malfunction, because they have been badly structured or because their condition is deteriorating. Being structured systems, computing systems are also subjected to this law.

Due to the fact that responsibilities are delegated to computing systems, their failure can have regrettable and even tragic consequences. The media

regularly reports on catastrophes affecting industrial, aerospace, avionics and rail transport systems. Firstly, we can take as example, the opening of Denver airport which was delayed for several months due to a failure in the luggage management system. A second example is the delay of the Ariane V project due to a control problem which had resulted in the failure of the rocket's first flight. Other examples include the recall of numerous cars by many of the major constructors due to potential anomalies of electronic systems. We also have in mind the failure of a space probe sent to Mars due to problems of non-homogeneity of measuring units (*inch* and *cm*). Finally, several patients were killed by failures of medical systems used for cancer treatments.

The transfer of responsibilities to computers and the fatalities due to their malfunctioning are two antagonistic facts. If economic reasons urge for the development of computing systems, their malfunctioning has to be avoided. In order to achieve this, we must first of all possess a good understanding of failure causes.

1.2.2 Failure Causes

1.2.2.1 Fault Notion

An electronic product is generally used in interaction with other products and human beings, which constitute its *functional environment*. For example, an electronic regulator controls the rotation speed of an engine used by a human operator to manufacture parts. The whole set of these partners (product and functional environment) is placed in a *non-functional environment* characterized, for example, by temperature and humidity.

The user generally notices the failure of a product during its operation, in the context of the application. All causes of failures are known as *faults*.

When a failure occurs, one first needs to find the origin of the malfunctioning. For instance, a failure of an engine's control system can be due to a breakdown affecting the electronic regulator (the product). The functional environment of the product can also cause failures. For example, the engine controlled by the previously mentioned regulator can itself have broken down or have been badly used by the human operator. Finally, the non-functional environment can provoke an application failure as well. For example, the use of an electronic system in an environment with a temperature which is too high or subject to radiation can provoke a deterioration in the technological means used to manufacture the product, and therefore can cause a failure.

The distinction between the notion of *failure* which qualifies an effect (the product does not provide the expected service) and the notion of *fault* is

important. First of all, if the failure is attached to the operation of the product, the fault could have been introduced at different stages of its life cycle. Moreover, whereas the failure is associated with the product, the fault can be attributed to the product itself, to the human operator, or to other external objects.

Several fault classifications are possible, according to the stage at which the fault occurs (*when*), the actual object affected (*where*) or the agent who is at the origin of this fault (*who*). We note lastly that it is often very difficult to identify precisely the exact cause of the failure: the precise nature of the problem, the moment it occurred, the object affected. This fault notion is therefore relative to the *means of investigation* used.

1.2.2.2 When

All products come from a more or less complex process which transforms a need into a usable product. The succession of the various stages of the life of a product is known as a *life cycle*, starting from expressing a need which is effectively the birth of the product, to the end of the mission. The most significant stages of this cycle are:

- the *requirement expression* which describes the expectations of potential clients: for example, the creation of a landing system without visibility is motivated by the need to ensure the continuity of the air transport service;
- the *specification* which defines the functionality of the product to be created: for example, the specification of a landing system without visibility expresses the relationships between the data elements provided by the environment via the sensors (radar, altimeter, gyroscope, etc.), and the orders transmitted to the actuators (engines, jacks, etc.);
- the *design* and the *realization* which lead to a solution proposed to handle a problem stated at the preceding stage, and to refine it until an expression using hardware and software technology is obtained: for example, an automatic landing system will be created as a software executed on a hardware platform using one or more electronic boards;
- the *production*, or *manufacturing*, which consists in reproducing the first product in several copies before they are put on the market;
- and finally the *useful life*, or *operation*, providing services to the user of the product.

Unfortunately, during each stage, faults can be introduced by the diverse intervening human beings or by the tools or technologies they use. In addition, these various and diverse faults have a tendency to accumulate, making their handling even more difficult, and hence the failure risk higher.

1.2.2.3 Where

In this book, faults are classified according to the product which is at the center of our investigations. We define three fault categories:

- the *internal faults*, attached to the product, which are divided into two sub-groups:
 - the *functional faults*, or *creation faults*, committed by humans or by their tools during the stages of specification, design/realization and production; for example, a software designer has badly translated the design model features by means of programming language statements;
 - the *technological faults*, also called *breakdowns*, affecting the execution means: for example, an internal connection in an ageing electronic circuit has been cut off;
- the *external faults*, or *disturbances*, arising from the product's environment; for example, a heavy ion modifies a value '0' into a value '1' in a memory of a control system embedded in a satellite.

1.2.2.4 Who

Humans constitute a fundamental source of faults. First of all, as a product user (for example a pilot of a vehicle), the human operator is reputed for committing faults. Numerous catastrophes have been or are still caused by human faults. The media reports cases affecting air, rail and automobile control, and energy distribution systems. Secondly, as product designer or manufacturer, humans introduce faults into systems.

In order to remedy these faults, it is necessary to introduce methods and tools intended to assist (to guide and check) or simply substitute the humans. For instance, a compiler automatically translates the statements of a source program into a set of instructions of an executable program. In the same way, CAD (Computer Aided Design) tools make the electronic circuit design easier, and consequently more dependable. However, let us note that these means are themselves issued from a human activity of creation. Thus, used methods and tools can also be affected by faults. Moreover, the product designer, again a human, can also commit faults by badly handling correct methods and design tools which he/she disposes of.

Finally, even if all these preceding factors of human faults have been mastered, the hardware equipment which processes the software application is still subjected to degradation phenomena: natural component ageing or external aggression due to high temperature, electromagnetic fields or space radiation. These perturbations affect the hardware platform, producing faults during the operation of the product.

1.2.2.5 Fault Analysis

Whatever its origin, a fault is often difficult to predict and identify. It can be studied in probabilistic terms, using experimental data issued from already manufactured products and statistical laws, such as the probability that an electrical component is affected by a single breakdown (a stuck-at, a short-circuit, etc.). Faults can also be analyzed by examining their effects. A fault leads to a failure damaging the service delivered by a product according to a transformation mechanism which creates internal *errors* and propagates these errors to the outputs (by contamination) through the internal structure of the product. It is not always easy to analyze this mechanism. When someone uses a product in a particular environment, it is difficult to precise if the failure results from unacceptable characteristics of the environment (excessive temperature or high radiation, etc.) or because the product is incapable of supporting such an environment. Finally, the cumulative character of faults again complicates the analysis: several quite different faults produced at different stages of the life cycle can lead to a same failure.

Faults, errors and failures constitute *dependability impairments* which must be well understood. The previous comments do not imply that the fight against faults and their effects is lost from the start. Of course, it is a difficult challenge, and several parameters can cancel out the efforts made to handle faults. Numerous protection methods and techniques exist, but they have to be employed together with competence and in the appropriate manner. These means are regrouped in the scientific discipline known as *dependability*.

1.2.3 Taking Faults into Account

The destructive phenomena due to faults and their effects cannot be ignored. System designers, manufacturers and users should be totally aware of this reality and take on their responsibilities. This implies integrating the fault notion at the very first stages of specification and design, then all throughout the product's lifetime.

For a long time, the approach considered in industrial projects consisted first of all in designing and creating a product from essentially functional specifications, then integrating dependability criteria at the end of the project. This approach is undesirable for two fundamental reasons: it is *expensive* and *inefficient*.

First, this approach is *expensive* because faults are not considered as soon as they are introduced. Their handling becomes more and more expensive during the following stages. In particular, if they are not corrected accordingly, they can lead to failures long after the product's commercialization. This impedes the product's appeal. The strongly

increasing character of this financial phenomenon is often quoted. Millions of euros and dollars are wasted by the loss of expensive systems (such as the probes sent to Mars), the cost of testing and repair, and the consecutive loss of the market and of clients trust (for example, due to vehicles being called back by car manufacturers).

This approach, which consists in considering a posteriori the dependability requirements, is *inefficient* because the good functional, methodological, and structural choices avoiding or reducing the appearance of failures, should be made right from the very beginning of the life cycle. In addition, all late fault elimination is more difficult. We estimate that in the electronic industry, the cost of fault correction increases by a factor of 10 at each stage of production: for example, component manufacturing, insertion of components on printed boards, assembling boards on racks, etc.

The only suitable approach consists in taking all the product's parameters into account from the very first specification stages: the functional parameters (product behavior) as well as the non-functional parameters which are relevant to the dependability. In particular, this implies being familiar with the environment in which the product is immersed: the functional environment with which it communicates (this could be a process or a human interlocutor), and with the non-functional environment (temperature, humidity, vibrations, etc.).

The principal handicap of such an approach is, of course, the additional constraints which are made obvious early on. These constraints increase the cost in the short term and are difficult to express in many projects. We should note that often the economic excuse results from a false calculation: as we have already emphasized, on a medium or long term basis, the cost of a badly studied product from a dependability point of view could turn out to be exorbitant.

A primary reason that incites engineers to postpone the treatment of faults until the end of their project is due to the fact that they badly manage this aspect. The traditional training schemes mainly lead students to propose solutions to problems, asking the trainer to respond to the question:

Is the solution I am proposing correct?

Furthermore, designers, like their employers, often measure the results of their work according to the *quantity* of the product provided and not to its *quality*. For example, during software programming step, it is easy to measure the engineer's productivity by the number of lines of code produced. In this case, the time taken to avoid the introduction of faults into the program whether immediately or in the future (by its readability feature for example) is not taken into consideration. And as long as this activity is considered as lost time, no progress will be made.

Finally, the designer has a tendency to hide the existence of faults and the time he/she spent to avoid and to correct them, as this is considered as embarrassing (notion of *professional negligence*).

1.2.4 Definitions of Dependability

Two points of view are concerned by the definition of the dependability notion:

- dependability as attributes of systems,
- dependability as a science.

Due to the role and responsibilities attached to them, computing systems have to be characterized by their capacity to deliver the services for which they have been designed. They should not fail. This ability is expressed by attributes defining *dependability* of these systems.

To obtain this result, that is the actual ability not to fail, engineers developing these products have to use protective means throughout the whole development cycle. These methods, techniques and tools will be regrouped in a scientific domain also known as *dependability*.

1.2.4.1 Product Dependability

The *dependability* of a product has to be considered as one of its specification attributes, as well as the purely functional or economic requirements. The now classic definition given by *J-C. Laprie* describes the large scope of this term but also its precise objectives:

Dependability is that property of a computer system such that reliance can justifiably be placed on the service it delivers.

The service delivered by the product corresponds to the function which it performs during its operation.

Several scientific criteria allow the trust placed in a product to be justified, such as the *reliability*, the *availability*, the *maintainability*, the *testability*, the *safety* and the *security*. These criteria are called *dependability attributes*.

Reliability is attached to the study and the evaluation of the aptitude of a product to ensure its mission in a specified environment. This criterion is therefore concerned by the durability of the service delivered over time. *Maintainability* and *testability* attributes concern products on which it is possible to act in order: i) to avoid the introduction of faults, ii) when faults occurs, to detect, to localize and to correct them. *Safety* is specific to dangerous effects of failing products. *Availability* measures the aptitude of a product to function correctly, by integrating protection mechanisms

(maintenance or tolerance). This criterion only differs from the reliability criterion when these protective mechanisms are used. Finally, *security* groups together *confidentiality* (non-occurrence of unauthorized disclosure of information) and *integrity* (non-occurrence of improper alterations of information). This last security criterion is not considered in this book.

The obtaining of dependable products implies capabilities, means and tools which act on certain of the product's attributes. This has given birth to a relatively new discipline which involves the analysis and the prevention of product failures. This discipline has led to new design and production methods, which, when joined to the improvement of the technology used for the realization, allow the creation of new products which have better dependability.

1.2.4.2 Dependability Science

As a science, *dependability* proposes a global approach to study and design products which provide a 'justified trust' in the service they deliver. This approach attempts to unify the two processes which, on one hand guarantees the achievement of a product which functions correctly and, on the other hand, guarantees that the product will function correctly during its whole active life in the environment in which it is being used.

Implied in several activities, such as the *design*, the *production*, and the *operation*, dependability is a very active discipline which gives place to scientific studies in many research laboratories and which leads to numerous industrial applications. This theme can be qualified as orthogonal, as it is used in many different domains:

- hardware and software systems considered in this book, but also the mechanical systems, for example, redundancy techniques such as the spare wheel of a car or the duplication of a braking system,
- biological systems, as nature offers an impressive range of efficient redundancy techniques, like for example the duplication of numerous organs (lungs, kidneys, etc.), which improve human dependability,
- or else socio-economic systems.

1.3 MEANS OF DEPENDABILITY

1.3.1 Evolution

Needs for dependability are not new. As soon as humans began to manufacture weapons and tools with stone, wood and bone, they quickly understood the need to produce solid and durable objects. The financial and

commercial needs appeared later on. This notion of *reliability* was structured into a scientific discipline even later according to the historic scale. The current explosion of digital electronics and its applications to computing have extended and amplified studies on reliability. Today, techniques allow the products to survive longer (high reliability) in isolated and/or aggressive environments. The aerospace and nuclear domains have greatly contributed to this development. To simplify, historic evolution reveals three main steps:

- technological improvement,
- mastering the development process,
- managing external relationships.

1.3.1.1 Technological Improvement

This is the first step towards the improvement of dependability. This concerns the implementation means and aims at mastering the technological faults. In the 1950s, at the beginning of computer systems, the first logical electronic components used (relays, then vacuum tubes, and then elementary transistors) had a very short average lifetime: the famous computer ENIAC (Electronic Numerical Integrator And Computer) of the 40's had 18800 vacuum tubes, 6000 switches, 10 thousands of passive components and offered an average life time of half an hour! Throughout successive technological generations, SSI, LSI, VLSI, etc., the MOS integrated circuits (pMOS, then nMOS, and finally CMOS) became more and more complex, and economical. At the same time, their performance and their reliability increased in a spectacular manner.

1.3.1.2 Mastering the Development Process

The continuous improvement of the reliability of physical components has allowed developing products more and more complex in quantitative terms (number of elementary components and of offered services) and also qualitative terms (sophistication of the provided services). However, the mastering of their specification, design and production stages is becoming increasingly difficult, implying the possible occurrence of numerous faults. Since the development of the very first computing systems, a lot of time has been spent trying to find out how to master the effects of creation faults. In order to mask or detect and correct errors coming from the product's operation, *redundancy* techniques were proposed, implying more hardware and/or more software. This approach was first used in the transmissions domain with error detecting and correcting codes, to stamp out the effects of the parasitic aggressions that disrupt the media of transmission.

Afterwards, certain error detecting and correcting codes defined for transmissions were adapted and implemented into computer units such as storage units. The new needs for logical and/or arithmetic treatment circuits in aerospace projects, industrial command-controls, and communications, have led to specific new redundant hardware and software techniques.

Following this, other complementary techniques have been introduced to limit the creation of faults. Their first objective is to master the product during its creation by using verification methods of intermediate models. Thus, a formal product specification allows the immediate detection (without waiting until the final stage of implementing the product) of cases of inconsistency or incompleteness which unavoidably result in product failures. Then, noticing that the creation faults are issued from the creation activity, the mastering of the development process itself was investigated. For example, in the software domain, the use of a programming style constrains the manner of programming and aims at avoiding certain types of faults induced by the difficulty in creating and modifying the program under development.

1.3.1.3 Managing External Relationships

Finally, interest has been extended to the *external faults* associated with the interactions between the product and its environment. Nowadays, these faults are in greater and greater, as digital systems have more and more external (with their environment: processes, users, etc.) and internal (other digital systems) relationships. For example, the task of regulating airplane engines is included in the flight control software, which itself is integrated into an avionics system. The relationships between systems have become more complex than just an inclusion. Systems interact between themselves and have to cooperate in order to fulfill a more global mission. For example, by placing intelligence in the form of software and electronics in a sensor, this device is transformed into a system which dialogues with the control system to inform it of certain events, whereas before it was purely a slave. Numerous faults can also result now from the interaction of the product with the human user, due to the complexity of these interactions.

The environment may also disturb the system operation, due to non-functional aggressions: temperature, heavy ion, electromagnetic fields, etc. moreover, the products are often themselves the source of disruptions of other systems. For example, a mobile phone produces electromagnetic waves which can disrupt avionics systems. For this reason the use of these telephones is banned during flights.

1.3.2 Means

The means offered by dependability science are numerous and varied. The related methods and their associated tools are traditionally organized into four groups: *fault prevention*, *fault removal*, *fault tolerance* and *fault forecasting*.

- ***Fault Prevention*** aims at reducing the creation or appearance of faults during the life cycle of a product.
- ***Fault Removal*** aims at detecting and eliminating existing faults.
- ***Fault Tolerance*** aims at guaranteeing the service provided by the product despite the presence or appearance of faults.
- ***Fault Forecasting*** aims at estimating the presence of faults (number and seriousness).

These four classes of means are complementary and should be considered jointly during the development of a product. For example, the implementation of tolerance techniques to handle faults in operation is not efficient if fault prevention and removal techniques have not been applied during the development process. Indeed, the tolerance mechanisms assume certain hypotheses on the faults tolerated. For instance, replicated modules tolerate hardware faults due to ageing, as such faults are supposed to concern one module only at the same time. On the contrary, a design fault can infect all the modules, making this technique inefficient. Consequently, this design fault has to be prevented or removed before operation. In the same manner, the means of fault removal suppose the prior use of fault prevention techniques. Effectively, fault elimination requires their detection and then their localization. These operations are expensive in technological means and in time. They become impractical if the number of faults treated is high.

In addition, prevention, removal, and tolerance techniques have to be applied efficiently and pertinently, as they are very expensive. Let us take for example the techniques of fault removal. Certain electronics industries estimate that 50% of the manufacturing cost is due to the development and the application of test sequences aimed at detecting faults. In the software domain, specialists say that 30% of the development cost is implied by the checking means of the developed applications.

1.4 SUMMARY

Introduced in the preceding sections, the main characteristics of the dependability of computing systems are summarized in *Figure 1.1*. They are structured into three groups:

- *Impairments* which involve faults and their progressive transformation and propagation through the product structure as errors and failures, and finally their external consequences on the mission.
- *Attributes* which provide designers and users with criteria (reliability, availability, maintainability/testability, safety, security) allowing to specify the expected dependability and to estimate the actual dependability of a product thanks to fault forecasting tools.
- *Means* used by the developers in order to provide the final product with the required dependability level; these techniques are organized into four sub-groups: fault prevention, fault removal, fault tolerance, and fault forecasting.

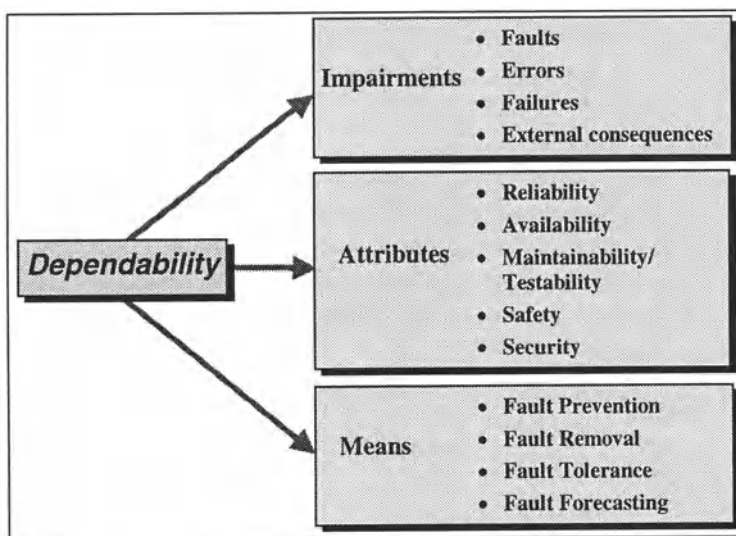


Figure 1.1. Dependability characteristics

The organization of the book is based on these groups. A first part deals with the impairments and analyzes all basic notions of destructive mechanisms. A second part provides the reader with a global overview of the protective means that can be used to increase the dependability of a product, and the attributes used to measure this dependability. A technical chapter is dedicated to redundancy which plays a major role in all protective techniques. Parts three and four aim at mastering the most important techniques of fault avoidance and fault tolerance means.

Chapter 2

General Context

In this chapter, we define a general context in which the dependability concepts can easily be introduced. We consider hardware and software products, created and designed for applications embedded in a given environment. This general scheme corresponds to a wide and significant range of real situations. In section 2.1, we firstly present the product in the final context of its application. Then, in section 2.2, we note the principal stages which led from the initial requirements to an operational application, according to a simple linear life cycle. We follow this life cycle in section 2.3 by tackling the modeling of a product as a system. To finish, in section 2.4, a simple example of a drinks distributor illustrates the notions presented.

2.1 APPLICATION CONTEXT

The class of applications considered is represented by *Figure 2.1*. We distinguish three parts:

- The **product** is a physical entity destined to satisfy a need of one or several *users*. The products considered in this book are implemented by hardware and software technologies.
- The **user** is the grouping of entities interacting functionally with the product via its inputs/outputs. The user is also called the **functional environment**. A user may also be another product such as an industrial process (e.g. an engine) or a human operator using the product.
- The **product-user** couple is immersed in the **non-functional environment** often simply called **environment**. This notion refers to external entities

which have an impact on the product's behavior without direct action on its inputs. It is defined by a set of non-functional parameters such as temperature, humidity, vibrations etc.

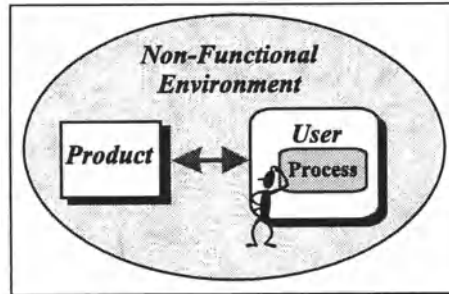


Figure 2.1. Application class

An application domain typical of this model concerns the process control, which associates a regulator (the product) with a user constituted of a process (the system controlled by the regulator) and a human operator. We remark that the process, functional partner of the product, is very often itself a manufactured system (e.g., an engine or an electric heating system).

Only the product is the object of our interest in this book, relatively to the dependability. However, the product dependability also depends on the user and the non-functional environment. Take for example an automobile ABS system. The product is an electronic regulator destined to satisfy a need: to avoid the blocking of the wheels when the braking is too strong. This regulator interacts with the user composed of two elements: the driver and a physical device constituted by the hydraulic braking system. The process sends the speed of the wheel's rotation to the regulator which then computes the control signals to be sent to the braking system. The non-functional environment is defined by a large number of parameters characterizing the vehicle, the road, etc. For instance, the non-functional environment may take into account electromagnetic radiation coming from diverse external sources (radar, mobile telephone, engine, electrical atmospheric parasite, etc.) as they may act on the electronic product.

Integrated into its environment, the product must ensure a *mission* which specifies the product's objective, i.e. the *function* to be performed and its *duration*.

- The *function* defines what the product is intended for and justifies its existence; it expresses the interaction with the user to whom the product is connected, that is to say the relation between the inputs and outputs of the product.

- The *duration* or *operational lifetime* of the mission varies according to the application: this can be a few seconds (missile flight), hours (airplane flight), months (space mission) or years (industrial regulation system).

For example, an ABS system can be defined as a function which reduces the pressure of the hydraulic braking system by a given amount via actuators when the speed of the wheel's rotation is inferior to a certain value depending on the vehicle's speed.

We call *delivered service* the product's real behavior when placed in its applicative environment. Thus, the *function* is the desired service when the *delivered service* corresponds to the one effectively obtained.

Product Examples

The following examples and their variants described afterwards will illustrate the notions and techniques introduced in the following chapters.

Temperature Regulator. The first product is a boiler controller. For example, it is constituted of a micro-controller which executes a regulation program. The controller receives sampled information about the temperature of the heated recipient. It then elaborates the reactions on the heating system by using an algorithm which takes into account a behavioral model of the recipient. These reactions act on an electro-valve controlling the combustible flow. The significant elements are:

- *Product*: Regulator + temperature sensor + electro-valve.
- *User*: Heated recipient + heating combustible + the human operator who fixes the regulation instructions (desired temperature).
- *Non-functional environment*: external temperature, corrosive gases, pressure, etc.

Drinks distributor. The drinks distributor has electromechanical and electronic parts. For example, the distributor is made up of an automaton for the choice of drinks, another one managing the money, and a third one for the drink's distribution. The functional environment includes the user as well as the service for supplying water, coffee cups, etc. We note that one can have a different point of view about this application by considering that the product is the purely logical part of the distributor (a set of interconnected automata) and that the functional environment includes the user, the electromechanical parts and the management of the fluids, ingredients, and so on. In this case, the objects are defined in the following way:

- *Product*: the logical electronic part of the control system (Programmable Logic Controller - PLC -, micro-controller running a program, Field Programmable Gate Array - FPGA -, etc.).

- *User*: the process which includes the electromechanical parts and the constituents of the drinks (cups, water, drink doses), and the human who uses the machine.

We can note that such a system makes a second human intervene: the operator responsible for recuperating the money and supplying the machine with drink doses. The product also has to interact with this user.

Arithmetic and Logic Unit. An Arithmetic and Logic Unit is a product interacting with a human operator (such as a pocket calculator) and/or an electronic process (circuit embedded in a computer). Both constitute the functional environment of the product. The non-functional environment can be characterized by the temperature which has important effects on electronic components. In particular, if this temperature is too high, the service delivered can be different from the expected function.

Stack. A stack is a product which stores data. It can write words in sequential order and read them in the reverse order of their writing. This memory unit is very useful in computing; it can be implemented either as a hardware component (specific electronic circuit) or a software product (offering the functions to write and read). We will consider two significant applications:

- the management of subprograms calls using a software stack to back-up the local variables and the calling subprogram return address,
- an interrupt management system to back-up the current application context (the values of some internal registers) when an interrupt occurs, to restore this context when the interrupt handler execution is completed.

In the first case, the functional environment is the executable program which calls the write (PUSH) and read (POP) functions.

In the second case, the environment is an electronic circuit which provokes the back-up (respectively recovery) of the context by PUSH orders (respectively POP orders) at the initialization of an interrupt treatment (respectively during its conclusion).

Industrial robot. As illustrated by *Figure 2.2*, an industrial robot is made up of several objects belonging to two parts:

- the *product* called the controller, which can be a control program running on a hardware platform,
- the *user* including the robot, the tools it uses (machine tool), the pieces it treats (manufacturing and assembly) and the human operator.

Other humans can also interact with this product: the maintenance agent and the instructor responsible for the training of the robot.

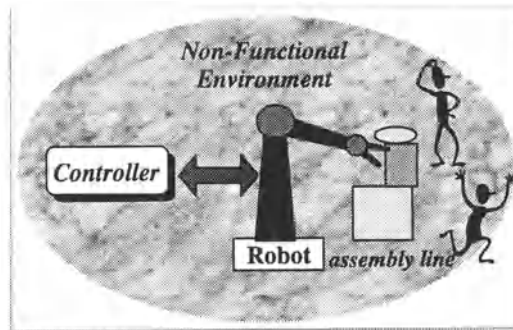


Figure 2.2. Industrial robot

2.2 LIFE CYCLE

2.2.1 Principles

As already pointed out in the introductory chapter, the failures which affect a product's mission arise from the faults which appear all throughout the life of this product. It is therefore necessary to first identify clearly the stages of the *life cycle* of a product in order to analyze the type of faults associated with them and then to apply the appropriate protective mechanisms.

Life cycle starts with the *expression of a need*, which defines the expectations of the future product's users. It gives an answer to the following question: *why* do they have a need for a product? This need is formalized by the *requirements* which justify the creation of a product. They are determined from future users by means of *requirement capture techniques*. This stage is not considered in our book. We start therefore the development of a product from its given requirements.

In order to facilitate our study, we consider a simplified cycle which has four *phases* (also called *stages* or *steps* here), as illustrated by Figure 2.3:

- the *specification* (or establishing of an initial *contract*) which defines the product to be created by the *expression of specifications*,
- the *design* which transforms the specifications into a *system* which is a priori an abstraction without physical reality,
- the *production* (also called *manufacturing* or *implementation*) which finally transforms the system into a real product using hardware and/or software technologies,

- the *operation* (or *useful life* or *utilization* or *exploitation*) which integrates the product into a given environment to execute a mission.

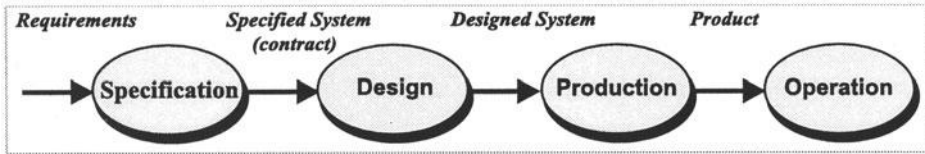


Figure 2.3. Simplified Life Cycle

The *development process* (or *development* or *creation process*) of a product groups together all the operations necessary to obtain the final product. So, in the general framework of our linear life cycle, this process regroupes the three phases: specification, design and production.

Other development process models break up or combine these phases. For example, the *spiral cycle* iterates the preceding phases by progressively taking into consideration diverse aspects of the requirements. However, these models do not question our explanations regarding to destructive and repair mechanisms that will be studied for each phase.

2.2.2 Specification

Written and signed by the *client* (the person who has proposed the project), the *designer* (the person who creates the product based on the needs expressed by the client), and sometimes the *user* (this word implies here the human who represents the functional environment of the future product), the *specification* formalizes the characteristics of the product to be created. Figure 2.4 symbolizes this relationship between the three partners.

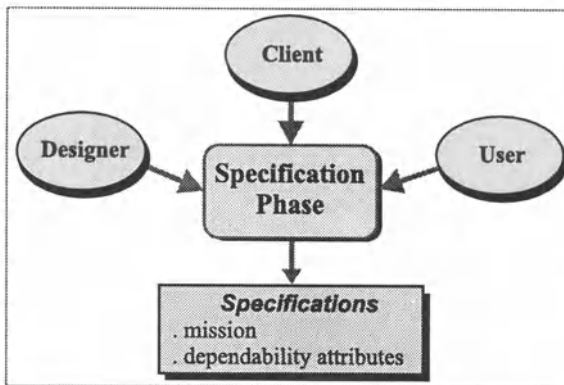


Figure 2.4. Initial Contract

From an expression of the requirements, generally written in natural language, this initial phase of the project establishes the formal or non-formal specifications constituting a *contract*. This contract defines two points:

- the *mission*, also called *functional characteristics*, of the product, that is to say the function or service to be delivered by the product and the duration of its operation,
- *non-functional characteristics*, dealing with product dependability requirements (according to attributes such as reliability, availability or safety) and constraints (such as temperature or radiation) of the non-functional environment of the future product.

The specifications include the formal definition of the relationships between the product and its environment (functional and non-functional). In particular, the role assigned to the product and the constraints of the environment for which this role is desired must be defined. For example, the contract defining a program can have to specify the constraints on its executive environment in order to guarantee a good functioning. If a user acquires a workstation which integrates this software, other constraints are expressed (the temperature of the room, etc.) to guarantee a good functioning of the hardware. Too often, non-functional environment parameters are missing or incomplete, leading to numerous ulterior problems.

In addition, the expected trust in the service delivered by a product, that is its dependability requirements, has to be quantified in terms of a set of *attributes* the values of which have to be specified. The effective values will then be established at the end of the development process. For example, we will specify the reliability of a certain circuit with a probability lower than 10^{-5} that the product will have a failure during a mission defined for 1000 hours. Then, this requirement will have to be compared with the estimated reliability of the final produced circuit.

Let us note that the two aspects of the non-functional characteristics are correlated. For instance, the mean time to first failure (reliability metrics) of an electronic component is strongly correlated with the environmental temperature. A required value for the mean time to first failure will not be guaranteed if the temperature is too high!

In numerous cases, the client and the user are the same physical person. Sometimes, the client is also the designer. However, these three partners (client, designer and user) will be voluntarily distinguished here because they correspond to different and sometimes antagonistic points of view. This is the case concerning the notion of *service delivered* which corresponds to a user vision of the functioning of a product. This vision has however to be accepted and understood in the same way by the three partners.

Unfortunately, very often the contract is not a formal notion implying the three partners: for most industrial products, the user who buys a product has not participated in the contract which has been established by the client (who imagined the product) and the designer (who created the product).

2.2.3 Design

2.2.3.1 Introduction

Design is a process transforming the specifications into a *system* which constitutes an abstraction of the future product. What distinguishes the modeling from the product itself is either its incapacity to execute itself (being a simple description), or the fact that it does not take into account the available execution means involved during the operational phase. For example, it is possible to represent and to simulate an application carrying out several tasks on a mono-processor without taking into account the distributed aspect of the execution means (the final product has to operate on a multi-processor network).

The product is rarely obtained directly from the specifications at one go. It results most often from a succession of linked stages, the number and the nature of which depend on the type of product considered and on the chosen design process. Each stage leads to the description of the system by a model. Thus, the more complex is the behavior described by the specifications, the more numerous are the stages to be carried out. Actually, each stage refines the results of the analysis of the previous stage (from a general design towards a detailed design). In addition, this refinement process depends on the implantation technologies chosen according to performance criteria (which may favor electronics) or maintainability (which may lead to the development of a software model) or others.

Sometimes, a step of *realization* providing a model of the product concludes the design phase. This step produces a model which takes the features of the execution technology into account. This could provide, for example, an executable program when software technology is employed, or a CMOS technology circuit for a hardware implantation. This step is integrated into the design phase in order to facilitate our study. It should be noted that this inclusion is justified, as the last phases of design using technological means are often automated. For example, a program written in a programming language is a model for which the implementation, that is to say the executable code, is obtained thanks to a compiler.

We are now going to succinctly present the processes of design used for hardware (section 2.2.3.2) and then software (section 2.2.3.3) systems, and to finally conclude on their similarities (section 2.2.3.4).

2.2.3.2 Hardware Design

The design process of an electronic circuit can be organized into three successive levels implying different models and methods (*Figure 2.5*).

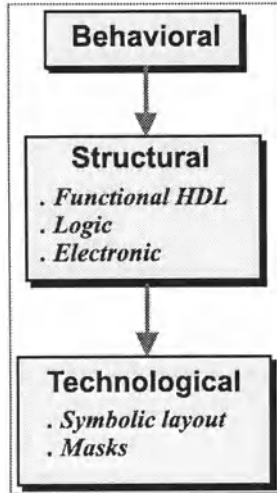


Figure 2.5. Creation steps of an integrated circuit

- The *behavioral level* (or *system level*) expresses the global functioning of a product without any knowledge of its structure. For example, the used model can be a finite state machine or a set of input/output sequences. If no formal specification has already been provided, this level generally involves a first stage of formalization of the system.
- The *structural level*, on the contrary, is supported by a structural model: the system is structured into interconnected entities called *modules*. Here we can distinguish three intermediary levels:
 - The *functional hardware oriented level HDL* (*Hardware Description Level*), such as VHDL (VHSIC Hardware Description Language, a IEEE 1076.1 industry-standard) and Verilog, which are the principal languages used nowadays in the industry); this level uses reference hardware modules (Arithmetic & Logic Unit, registers, counters, memories etc.);
 - The *logic level*, which reveals the system as a structure of interconnected elementary logical modules: gates (AND, OR, NAND, NOR, XOR, etc.), flip-flops and various logic networks; the resulting structure appears as a *netlist* of interconnected components;

- The *electronic level*, which employs transistors (switching elements) organized into networks (static and dynamic logical networks, etc.).
- The *technological level* (also called *layout level*) expresses the physical reality of the integrated circuit in the form of a layout, that is to say a topology of several layers. For example, the MOS technology uses P & N diffusion wells, polysilicon gates, metal 1, metal 2, etc. This level breaks itself down into two sub-levels:
 - The *symbolic level* where the different technological layers are represented by color lines, such as William's classical N-MOS *stick diagram* popularized by the Mead & Conway's book: green = diffusion, yellow = ionic implantation, red = polysilicon, blue = metal.
 - The *mask design level* which expresses the real topology of the different masks necessary to the production of the integrated circuit.

In the case of an electronic system with supply circuits and other components associated with the electronics (sensors, actuators, magnetic or optical disks, signal couplers, etc.), the design involves the preceding levels (behavioral, functional, structural, technological) for each electronic part, but also carries out integration and assembly stages, leading to several PCBs (Printed Circuit Boards) interconnected by various connection devices.

2.2.3.3 Software Design

As for hardware systems, the *behavioral model* of a software system is formalized if this has not already been carried out during the specification stage. Following this, a *structuring of the system* by breaking it down into sub-systems is established (preliminary design), then refined (detailed design). According to the type of product, the control relationships between the sub-systems are:

- *Sequential relationships*: sub-systems offering services called for in sequence, eventually in a repetitive (`loop`) or conditioned (`if ... then ... else ...`) manner,
- *Parallel relationships*: sub-systems are cooperative and/or competitive in an independent way (parallelism) or being constrained by precedence relationships (synchronization).

The design process is reiterated on each sub-system. The different expression models, resulting from each design stage, are expressed by notations (for example HOOD, UML or programming languages).

This work finishes by a programming stage which transforms the last design model into a program. This operation is made easier due to the automatic generation of a part of the source code by development tools.

Finally, the executable program is obtained using a *compiler* and a *linker*. It makes use of the services offered by the execution means: a *microprocessor* interprets the basic processing services and an *operating system* offers more abstract management services (management of inputs/outputs, tasks, etc.).

2.2.3.4 Similarities

The presentations, carrying out separately the two implementation technologies in the two preceding sections, show a similarity in the process and the means used. Three *design levels* are implied: behavioral level, structural level and technological level.

- First of all, a *behavioral level* of the description of the product formalizes the specification in both cases). The models used are the same: automata, *Petri Nets*, *StateCharts*, etc. This modeling allows the designer to understand what the product needs to do, and to detect information lacks and inconsistencies. For example, in the case of a Petri Nets model, the designer can detect certain undesirable deadlock situations. It should be noted that this work needs to be carried out during the specification stage. This is sometimes impossible when the contract is written in natural language.
- The *structural level* is also present in both cases. The module (or component) notion represents the break down of a system into interconnected sub-systems according to links of type ‘is composed of’ (*composition relationships*) and ‘calls to’ (*service relationships*). At the beginning, these modules are abstract elements, and then they are materialized at a logical level (HDL modules or subprograms) to progressively reach the technological level (transistor networks or programming language statements).
- The *technological level* concerns the ‘materialization’ of the design modeling to obtain an executable system. In the case of software, this involves the translation of the features of the programming language used. Two sub-levels exist jointly for the two technologies:
 - The *symbolic level* which gives an abstract view of the execution means. We have already quoted the example of the stick diagram in electronics. We can also quote the *abstract machine* associated with the software programming language. For example, the use of subprograms implies the use of a stack which allows the storage of return addresses by the caller (stack machine).

- The *physical level* which implements the concepts of the previous abstract view. For example, according to the hardware used, the abstract notion of stack necessary for the management of subprograms will be directly offered by the microprocessor (PUSH and POP instructions) or should be simulated by software features (stack implemented as an array and a 'top of the stack' pointer). In electronic technology, the integration means from the layout level to an integrated circuit implies numerous phases which depend greatly on the technological processes used.

To conclude, it should be noted that in numerous cases the first design process phases are independent of the final implementation means (software or hardware). Several phases of the classical *functional* approaches (SA-RT) or, more recent *object* approaches (UML), can be applied both to hardware and software products.

2.2.4 Production

The *production* stage ensures the physical realization of a manufactured product having already been designed. Industrial constraints such as standardization, productivity and quality influence this phase. The actual means are very varied according to whether we are creating electronic, mechanical, electromechanical or even software equipment.

In the case of electronic systems, it would involve buying and/or manufacturing passive or active components (Standard components, ASIC, or Full Custom Integrated Circuits) or even circuits integrating several levels of functionality and power (like the System-On-Silicon - SOS), placing them on printed circuits, interconnecting the cards by diverse connection means, mounting these cards in racks, conditioning and packaging the final product. The system obtained at the end of the design phase will probably be a little bit modified in order to satisfy these production constraints. For example, if a system model is too complex to be economically designed by a unique integrated circuit, it is necessary to break it down into interconnected sub-modules, which adds a supplementary stage. The reuse of available components can also modify the result.

In the case of software, the implementation and adaptation of the product (a program) in its final context must take into account the executive environment: hardware platform (microprocessor, input/output unit, etc.) and operating system (real-time kernel, drivers, etc.). Frequently, this activity has been handled at the last stage of the design. The production of the software is essentially the copying of files onto varied media supports (ROM memory,

CD ROM, etc.) or even the transmission of data onto a distribution network (specialized lines, local network, Internet, etc.).

2.2.5 Operation

Useful stage in the life cycle of a manufactured product, the operation of the product is the outcome of the creation stages. The product is placed in interaction with the user in order to execute the defined mission in its final environment. The use duration is generally longer than the creation stages. It implies relationships with humans (users, and maintenance agents).

The *repairable products* integrate another stage linked to the operational phase. Actions processed on the product structure during its useful life are named by the generic term of *maintenance*. The first goal of this stage is to proceed to the operations of fault detection and removal necessary to reach the satisfaction of dependability requirements (*preventive* and *corrective maintenance*). ISO 8204 defines *corrective actions* as actions taken to eliminate the causes of an existing non-conformity, defect or other undesirable situation in order to prevent recurrence. In addition, economic competition leads to the adding of supplementary functionalities or to the improvement of the product's qualitative execution performances, ergonomic properties, etc. of existing functionality. Hence, the specification may be modified, leading to the design of a new version of the initial system and an adaptation of the production process (*evolutive maintenance*).

From the product's designer's point of view, the maintenance phase is very important although often underestimated. In effect, numerous products have a creation which lasts 3 to 5 years and a life duration of 20 to 50 years (it is the case of avionics products). This means that the maintenance costs charged to the designer can often be as expensive as development costs of the first version of the product.

2.3 PRODUCT MODEL

During the specification and design stages, the product is represented as a system by using different *modeling tools* also often called *languages* (when their semantics is formally defined) or *notations* (when their semantics is not formal). For instance, Petri nets, FSM (Finite State Machines), Ada and C languages are modeling tools. Elements of a modeling tool are called *features*.

A *model* is one instantiation (one use) of a modeling tool to express a specific system.

A **system** is a set of components or sub-systems that act together as a whole to achieve a given mission, that is to say a given *function* for a given *duration*.

The system reveals structural aspects and functional aspects introduced in the following section.

2.3.1 Product Structure and Functioning

As introduced in section 2.2, most designs of complex products lead to models structured into sub-systems called *components* interconnected by *logical links*. We come back to this notion in order to precise its meaning at a system level, independently from hardware and software aspects.

A **component**, also known as a **module** or as a **sub-system**, is an entity of a system which carries out a precise function.

The abstraction level of this notion is relative: a component is for example a part of an integrated circuit, a complete integrated circuit, a board, a subprogram, a task or package, but also a computer or a network server. In fact, a component is a system considered as a part of another system.

The **logical links** between the modules are of a very variable nature: electrical wires carrying logical signals (levels or pulses), media transporting messages, call mechanisms for subprograms with parameters passing or synchronization protocols for tasks in software, etc.

The modeling which defines a system as a set of connected components is the **structure** of the system.

As a module is a sub-system, the function which is associated with it is specified with a classic behavioral model such as a ‘logical expression’ or a ‘finite-state machine’. As for a system, the module’s external interface has to be defined by input and output variables. Its behavior also has to be clarified as the module is *reactive*: it reacts to the application of new input values by provoking an internal evolution, leading perhaps to new output values.

A module’s behavior is defined by **attributes** which characterize it at the considered level of abstraction. If we take the example of an elevator, the position of the cage is one of the attributes which can be useful when identifying its behavior. The values taken by these attributes define the **state** which is a characteristic property of a module at a given moment. For example, the states of the attribute *cage position* could be ‘ground floor’, ‘first floor’, ... , ‘twelfth floor’. Another attribute could be the movement having three states: ‘up’, ‘down’ and ‘stop’.

The **behavioral model** is therefore expressed by the changes of *state* of these attributes. For example, a program’s internal variable can be assigned

by values defined by a *type*. The actual value evolves during the running of the program, which illustrates the notion of state evolution. A circuit using several flip-flops circuits provides another example. The state of this circuit is therefore characterized by the binary configuration of these flip-flops (each one switched ‘on’ or ‘off’) which evolves with time.

The global model of a system structured into interconnected modules is called **structured-functional model** because it defines a system by its structure (the modules and their links) and the behavior of its components. This is illustrated by *Figure 2.6* and is found as much in hardware systems as in software systems.

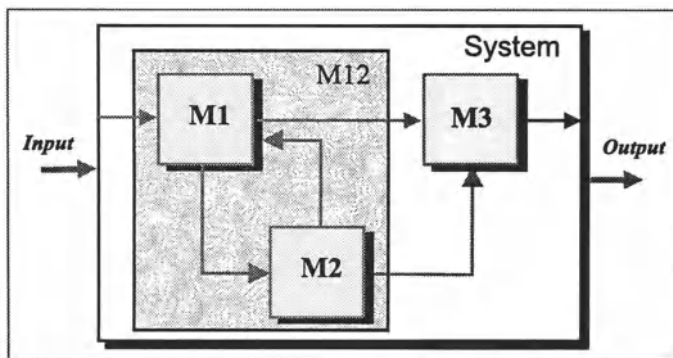


Figure 2.6. Product structuring

2.3.2 Hierarchy

In a hierarchical structure, a ‘father’ module breaks down into interconnected ‘children’ modules, which can themselves be broken down in a recursive manner until reaching ‘leaf’ modules which do not have children. As shown by *Figure 2.6*, the product breaks down into two children: *M12* and *M3* coupled by two links. *M12* breaks down into two interconnected children: *M1* and *M2*. Hence, the system is organized according to a tree structure, the leaves of which are non-structured modules. This hierarchy makes three levels appear: the global system, the modules *M12* and *M3*, then finally the modules *M1* and *M2*. Looking at this hierarchy therefore reveals three interconnected modules: *M1*, *M2* and *M3*.

This is a **compositional hierarchy** because the system can be uniquely represented by the leaves of the tree. The intermediate model *M12* is only present in order to give an abstract view of a part of the product. A second hierarchy, called **use hierarchy**, can be defined. It reveals the service relationships linking the components. For example, *M1* uses *M2* in order to provide a result sent to *M3*.

Let us consider two software examples illustrating these two hierarchy types. An Ada procedure may declare local procedures defining a *compositional hierarchy*. An Ada program may use external functions (such as input/output services), defining a *use hierarchy*.

The hierarchy notion leads us to express the relativity of what has been defined as ‘product’, ‘users’ at the beginning of this chapter. Let us consider the system shown in *Figure 2.7*: a controller is coupled to a process in a given non-functional environment. The controller product is structured as a regulation program (RP) running on a hardware platform (micro-controller and interfaces). According to the design level, we will consider as product the complete controller or the regulation software only.

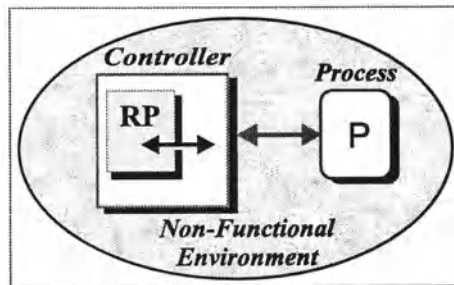


Figure 2.7. Insertion of a product into a control system

2.3.3 Examples

Example 2.1. Regulator

A temperature regulation system has been designed by means of three modules interconnected by logical Buses: an arithmetic and logic unit (ALU), a control unit, and a memory. The control unit takes its instructions and data from the memory and uses the ALU in order to calculate the control signals sent to the process. This product is an illustration of the structure of *Figure 2.6*: *M12* is constituted of the control unit (*M1*) and the ALU (*M2*), and *M3* is the memory. This is a compositional hierarchy.

Example 2.2. Software

The structuring notion applies also to software: a program calls subprograms which call other subprograms, etc. The calls are correlated by relationships:

- sequential (we execute *A* then *B*),
- conditional (if .. then .. else ..),

- repetitive (for and while).

Hence, these modules are linked and they exchange data (variables).

As for hardware systems, software programs reveal structure and hierarchy. *Figure 2.8* illustrates a compositional hierarchy corresponding to the following program:

```

begin
  A;
  B(X);
  while C(X) loop
    D(X);
  end loop;
  E(X);
end;
```

The link (1) represents the stream of control (*B* starts its execution after *A* has signaled its completion). The link (2) adds a stream of data representing the variable *X*. We should note that the component *C'* corresponds to a modeling of the statement 'while' in the programming language. This integrates the component *C* and the selection of the relationship (3 or 5) activated according to the Boolean result of the evaluation of *C'*.

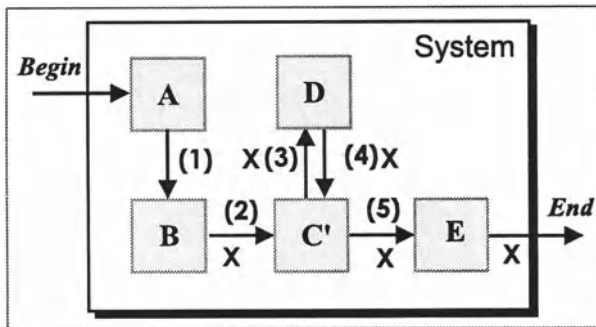


Figure 2.8 Structural model of a program

2.3.4 Refinement Process and Primitive Components

As mentioned in the previous section, the function of each module constituting the system is generally translated by a structure of sub-systems. The functioning of each of these sub-systems is therefore specified. This process is reiterated going down until reaching its 'primitive' components (that is to say which are not broken down). The transformation of a product behavior into a structure is obtained by successive stages leading to a hierarchy of modules. The intermediate modules have a specific behavior

which is put into work by the child modules. The modules at the lowest levels also have a behavior which is meant to be available.

We only consider systems whose behavior can be represented by *discrete models* which are the most used in computing. We put them into opposition to *continuous models* which are often those of the controlled processes. For example, the temperature or the speed of an engine may vary continuously.

Example 2.3. A software behavioral model

In section 2.3.3, we showed that a program constitutes a system's structured model. The bodies of primitive sub-systems (not broken down) express a behavioral model. This model has attributes such as the parameters and local or global variables of a subprogram. For example,

```

procedure Push(X in Element) is
begin
  Top := Top + 1;
  Stack(Top) := X;
end Push;

```

where `Top` and `Stack` are two variables external to this procedure (global variables).

These two variables can take different values which define the states of the behavior of `Push`. Thus, if `Top` is included in the range $[0 \dots 100]$, this variable then introduces 101 states. The behavior of the statement `Top := Top + 1;` is then described as a state change associated with the states of the variable `Top`. Thus, the execution of the procedure can be modeled as state changing.

2.4 LOGICAL PART OF A DRINKS DISTRIBUTOR

To conclude this chapter, we consider a simplified version of a drinks distributor. It has the following objects:

- *User*: the person who uses the distributor and the maintenance agent,
- *Client*: the person who launches the project of the distributor (who is probably the distributor dealer),
- *Designer*: the person who creates the distributor.

In this section, we provide an overview of the different stages of the life cycle of this product. Beforehand, let us examine the client's *expression of requirements*. It could be to earn money by selling drinks or to allow users to quench their thirst. The choice between these two needs is not insignificant,

since in the first case, a method of paying has to be specified and designed. In addition, the possibility of obtaining a drink without paying is therefore a failure of the product, which is not the case if the second need is considered.

2.4.1 Specifications

The specification defines the inputs and outputs of the system (interface) and their relationships (behavior). This last behavioral model must contain the checking that the chosen drink is available, and that the means for drink distribution (cup, liquid, sugar, spoon, etc.) functions correctly. The chosen drink should only be delivered if the user pays the asked amount. A cancel button allows the distribution process to be stopped (if the drink has not already been released) and the money to be given back.

We assume here that only one type of drink is delivered (*Drink-Delivery*) and that only one type of coins is accepted (*Coin*). The drink is served as soon as one presses on the *Selection* button and the change money is given back (*Change-Return*). If a selection is cancelled (*Cancel*), the money is then given back (*Change-Return*). The maintenance agent collects the money (by the command *Collect-Coins* which leads to the distribution of the money *Money-Left*) and fill up the doses of drinks (*Add-Doses*).

This global definition has to be formalized by expressing: the product's interface with the user (see *Figure 2.9*), and its expected behavior, here described by an automaton (see *Figure 2.10*).

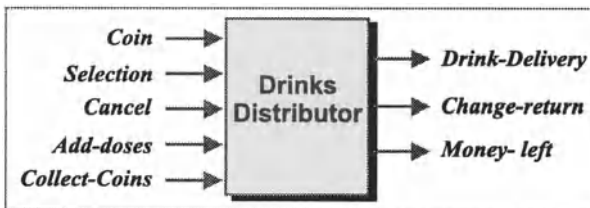


Figure 2.9. The Interface of a drinks distributor

This is obviously a specification model and not a design model, as it does not describe the means of putting the automaton into action and other necessary operations such as the addition of entered coins, the calculation of the money to give back, as well as the ways of detecting the introduction of coins, cancellation, the distribution of the chosen drink, the adding of doses, or even the money collecting by the maintenance agent. On the automaton in *Figure 2.10*, *Sum* stores the money introduced by the user, *Total* is the accumulation of money entered in the machine, *Amount* is the price of the drink and *Stock* is the number of available doses of drinks.

The behavioral specification expresses that the selection of a drink for a sum inferior to the amount required does not have an effect on its behavior. The user can then add more money or cancel. On the contrary, this situation could have been interpreted as having an effect equivalent to a cancellation. This example therefore illustrates the importance of the specification stage which implies choices which then influence the way of using the machine and which therefore necessitate a discussion with the client. In defining 'correct functioning', the specification will also be the fundamental way to state whether the product is failing or not when being used.

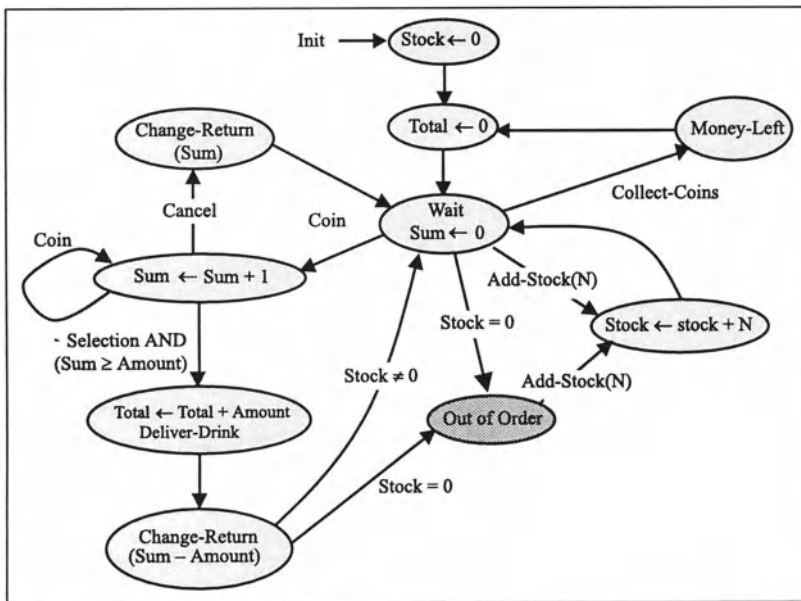


Figure 2.10. Behavior of a drinks distributor

2.4.2 Design

The formal *behavioral description* has been provided in the previous stage. The *structural modeling* has to reveal interconnected modules. Numerous refining choices are possible. *Figure 2.11* proposes a structure which makes two modules appear: a money device which manages the money, and a drink delivery module which manages the supplies.

The 'Money Manager' module accumulates the change provided, gives back this money when a valid cancellation is carried out and finally renders all the money contained when a demand is made by the maintenance agent in charge of the exploitation of the machine.

The ‘Drink Delivery’ module makes the drink available when the drink has been selected and when the drink is ‘selectable’, i.e. when a sufficient sum of money has been received. This last point justifies the link *Selectable-Drink* between the modules ‘Money Manager’ and ‘Drink Delivery’.

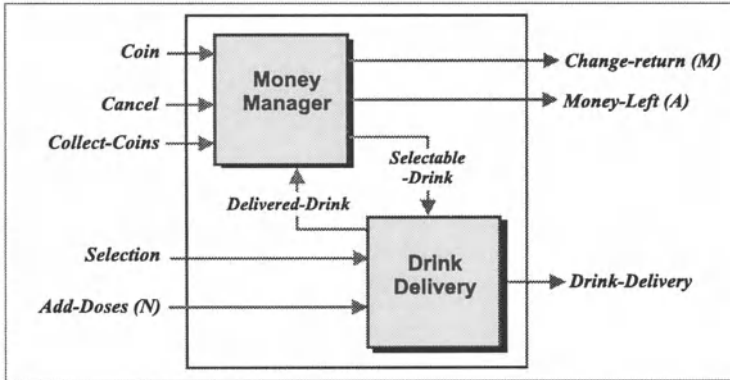


Figure 2.11. Structure of the distributor

In addition, when the drink has been served, the signal *Delivered-Drink* is sent to the module ‘Money Manager’ before it gives the change back. Then, each of these modules has to be studied separately and first of all its behavior specified, for example, by using an automaton or an algorithm.

We reach therefore the *technological level* concerning the execution means: hardware and software. The software will indeed be supported by electronic devices (micro-controller for example) and will have to communicate with mechanical devices (detection of the introduction of a coin for example) thanks to other electronic components (an interrupt sent to the micro-controller for example).

We will not refine this design here. However, we should note that the software part of the design requires two types of studies:

- At the *symbolic level*, giving an abstract view of the means: for instance, the two modules ‘Money Manager’ and ‘Drink Delivery’ will be implemented by associating a task with each module and by expressing their sequencing in relation to external events (*Coin* introduced, pressing on the *Cancel* button) and their synchronization (for example by *Delivered-Drink* signal).
- At the *physical level*, implementing the concepts of the preceding abstract view: for example, the occurrence of the *Cancel* event could be implemented by means of an interrupt sent to the micro-controller which

supports the execution of the software associated with the part which manages the money.

In the case of a uniquely hardware design, each module will be transformed into a specific integrated circuit. The communication between these two physical modules is ensured by a protocol based on the signals *Selectable-Drink* and *Delivered-Drink*.

This example shows again the number of choices that the engineer has to do, when designing a product.

2.4.3 Production

Inevitable adaptations of the product obtained at the end of the design stage will be necessary to comply with production constraints and standards. The production introduces specific notions such as the *cost* of the used materials (in particular the electronic components, but also the royalties on executive software or the graphical environments if such tools are used), the *assembly duration*, *time to deliver* the product, and finally the *yield* of the production line. It is clear that these constraints have to be considered a priori as criteria intervening in the design choices. The production concerns nonetheless specific capabilities. Thus, just as the engineer has dialogued with the client during the establishing of specifications, he/she has to do it with the people in charge of production during the design stages. The client intervenes again at this stage to establish the standard documents of assembly and use.

2.4.4 Operation

The final use requires certain complementary means and actions, such as the product's physical installation on the site of operation and the information and the training of the users when the product is more complex than a drinks distributor. It is necessary to resolve certain specific problems such as the connection to an electric power network and the distribution of water necessary to make the drinks.

Finally, it is necessary to define a maintenance policy in order to get the money, add drink doses, and to detect and repair all eventual functional anomalies, and also to improve the product performance and functionality.

Chapter 3

Failures and Faults

In this chapter, we begin the study of *impairments* to the dependability of a product with the analysis of *failures* and *faults*. From the observation of anomalies in the behavior of a product during its use, we define in section 3.1 the notion of *failure* relative to changes in the delivered service. In section 3.2, we identify and classify the various causes of failures, known as *faults*, according to several criteria. In section 3.3 we then explore the life cycle of hardware and software products, looking for the diverse faults which can appear. Some faults will be analyzed using the example of a drinks distributor in section 3.4. We conclude in section 3.5, providing a classification of faults, and assessing its interests and its limitations.

3.1 FAILURES

3.1.1 Definition

As stated in the introduction, experience shows that a certain number of issues can appear during the useful life of any product, just as well an automobile, as a television or a drink distributor. Functioning anomalies of a product are observed during its use in its application context. *Figure 3.1* shows some examples of incorrect behavior of drink distributors.

Firstly, we reckon a product presents a *failure* during its use if the *delivered service* does not conform to its *requirements*. This notion of failure is however ambiguous, as an expression of needs can lead to diverse interpretations, that is to say to diverse expectations of the service to be provided by the product. The expectations of people intervening (client, user,

designer or manufacturer) can vary somewhat. A client's or user's point of view in terms of requirements is not necessarily the same as that of the designer or the manufacturer!

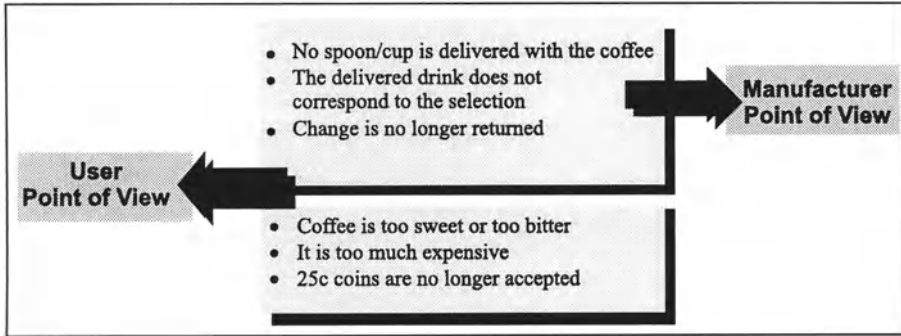


Figure 3.1. The distributor does not function correctly

A user of a drinks distributor, can effectively reckon the service he/she expects is not given because the spoon or the cup does not fall into place, or the machine takes the money and does not give back change, or else the coffee does not have enough sugar or is too bitter or even that it is too expensive. The manufacturer of the machine can have a different opinion, refusing as failures the last three cases (*Figure 3.1*). Moreover, the requirements interpreted by one of the partners of the initial contract can evolve as time goes on: the user tastes evolve.

What could be said about a distributor which only accepts $\frac{1}{4}$ \$ coins, whilst still providing a pleasant drink? From the client's point of view, the system responds to a need: to gain money honestly. Moreover, it does not satisfy the user who only has a $\frac{1}{2}$ \$ coin; this distributor does not respond to his/her needs at this time. This point of view may be acceptable if the three partners participated effectively in the initial contract. We often meet different situations for which the designer-client couple defines a product responding to a need, the user adhering to the contract latter on. This is the case with the majority of consumer goods: the user of a television did not participate in the contract which led to the design of the appliance!

We therefore understand that such a definition of failure is often subject to conflicts between the partners: this is a usual problem of user-seller relationships in everyday life, and all of us have had this experience.

This source of conflict is eliminated as soon as the service that has to be delivered is expressed in a complete and clear way, and if all the partners accept its definition. The needs are then translated in the form of *contractual service delivery*, established during the specification stage. The needs to be

considered integrate at the same time those of the client (“earn money from its product”, “low maintenance costs”, “reduced appliance costs”, etc.), those of the future users (“be able to use all types of coins”, etc.), and those of the designer and the manufacturer (“fairly high price in order to cover the costs implied by the product study”, etc.). The specification makes appear not only functional aspects interesting to the user (use of the distributor), the client and the designer (large stocks of coffee in the machine in order to reduce the refilling operations), but also non-functional aspects demanded by the user (for example the aesthetic appearance) or by the client (price of the distributor).

The first definition of a failure which will be retained afterwards in this book is the one proposed by J-C. Laprie in, “Dependability. Basic Concepts and Terminology”, Springer Verlag, 1992:

A failure occurs when the delivered service no longer complies with the specifications.

We should note that the notion of *mission*, which is the first aspect of the specification, integrates both functional aspects (what the product is intended for) and the length of its mission. In addition, the specifications define constraints on the non-functional environment.

These points appear in the definition provided by N. Leveson in “Safeware”, Addison-Wesley, 1995:

A failure is the non-performance or inability of the system or component to perform its intended function for a specified time under specified environmental conditions.

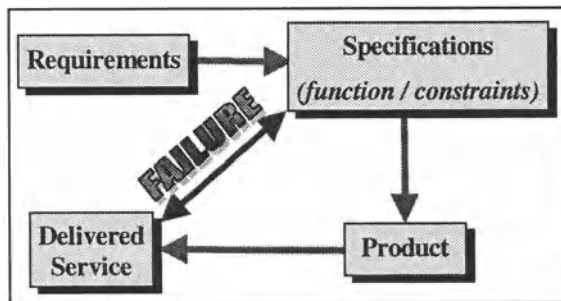


Figure 3.2. Failure definition

Thus, the failure stems from a comparison between the delivered service and the product’s functional specifications as represented in Figure 3.2, assuming the respects for non-functional constraints on the environment during the specified operational lifetime.

3.1.2 Characterization of Failures

The failures which can affect a product are various, and present multiple aspects. In this section, we introduce characteristics which are independent from the functionality of the product or the technology used. We could thus seek to develop general ways to treat the causes of failure in each class. These classes are often called *failure modes*. They represent abstract viewpoints on failures, independently of the particular system function.

We qualify the failures by three parameters a priori independent; each one can take two exclusive values:

- *static* opposed to *dynamic*,
- *persistent* opposed to *temporary*,
- *consistent* opposed to *inconsistent*.

Most of the system's behaviors reveal two aspects: a function elaborating the outputs from the inputs, and temporal constraints associated with the occurrence of outputs. This corresponds to the classic notion of static and dynamic response of a system. A *static failure*, also called a *value failure*, provokes a false result. The provided data are erroneous. A *dynamic failure*, also called here a *timing failure*, provokes a transient response which is incorrect, either too fast or too slow.

Consider a screen displaying information in a car. The displaying of wrong data illustrates a static failure. If the displaying task produces data whose values are correct but delayed, a dynamic failure occurs. For instance, data on the screen must be refreshed every 1/50 second whereas some of them are displayed after 1/10 second.

A product's mission takes place during a certain period. The second parameter corresponds thus to an observation of the product's behavior during time. A *persistent failure* alters the product's functioning for an important duration in comparison to the duration of the mission or definitely after a certain time. For example, let us consider a product regulating the temperature of a balloon. A systematic bad regulation due to a blockage of the electro-valve acting on the heating device is an example of persistent failure. On the contrary, a *temporary failure* presents a bad behavior at a certain moment during a short time. An erroneous control error of the electro-valve which appears at a given moment of the mission and will never happen again is an example of temporary failure.

The notion of consistency or inconsistency of a failure is relative to its external *perception* by several users. A *consistent failure* is perceived in the same way by all users. The failure is said to be an *inconsistent failure* in the opposite case. For example, several users of an office network complain

about not being able to access a printer from their workstation, whilst other users have no problem whatsoever. Inconsistent failures are also called *Byzantine failures*.

Figure 3.3 shows some examples of failures of the drinks distributor. Note that a failure can be static, temporary and consistent at the same time!

Static	The selection of coffee is no longer possible
Dynamic	The machine is too slow
Persistent	The machine has not been working since yesterday
Temporary	The machine sometimes refuses to function, in a random way
Consistent	All the users have the same opinion about the bad functioning of the machine (e.g. it does not give sugar to anyone)
Inconsistent	Some users only are unsatisfied
Static + Temporary + Consistent:	
The distribution of coffee is not possible for any user each morning between 08.00 to 09.00	

Figure 3.3. Examples of failures of the distributor

Other items can also be used to qualify some failures:

- **stopping failure** when the product's activity is perceived as no longer evolving, a constant value being delivered to the user,
- **omission failure** which is a particular case of the preceding definition when no values are delivered,
- **crash failure** which is a persistent omission failure (the system is definitely blocked).

Another way of classifying failures consists in considering the *seriousness* of their consequences on the application (user or environment). Actually, these consequences on the mission can greatly vary according to the application domains. We distinguish three main categories of failures:

- **benign failures** which can be ignored,
- **serious failures** which lead to a change in the mission with a certain cost,
- **catastrophic failures** (also referred to as *accidents*) which are not acceptable and stop the mission.

In the case of a temperature regulation system, the regulation can be badly assured (decrease of the production, the mission therefore not being stopped) or not assured (the mission is endangered with a loss of production, even a risk of accident). These consequences are known as *external*; they will be analyzed in Chapter 4.

3.2 FAULTS

Failures arise from a large number of causes which are informally named as *faults*. We find also in technical literature the terms *defect* for hardware technology and *bug* for software technology.

▮ A *fault* is an adjudged or hypothesized cause of a failure.

Even if fault seems to be a fuzzy notion, some important characteristics can be brought out.

3.2.1 Difficulties in Identifying the Causes of a Failure

The identification of a failure's cause (or causes as there could be several) is a difficult operation which requires important investigation means. Its complexity depends on the level of knowledge and observation that the analyzer has on the product, its functional and non-functional environment, but also on the process which has transformed a specification into a product.

According to the degree of accuracy of the observation, one can, by refining the analysis, come back to very long chain of causalities, sometimes without significance. For example, the cause of a failure of a hardware product could be found at the level of an integrated circuit or in a certain logical network of this integrated circuit or again a certain MOS transistor in the logical network and so on. Moreover, this refinement analysis can reveal potential multiple sources that are sometimes contradictory and the diagnosis of which is impossible or without interest. For example, if the previous MOS transistor is the cause of the failure, this is perhaps because it was abruptly blocked (this is then a *hardware fault*) or because the designer incorrectly dimensioned this transistor (this is then a different fault: a *human fault*). The initial cause can be due to the entropic hazard (e.g. component ageing), or a bad design which leads to a local overheating reducing the reliability, or else an excessive temperature in the final application.

In the same manner, the identification of the software fault which has provoked a failure of an executable program leads to the investigation of relationships which link the different subprograms making up the software.

This analysis will be more or less refined according to the degree of precision required for the fault's localization and the investigation means used (e.g. the observation of the variables). Furthermore, the software necessarily has relationships with an *operating system* and other software tools which manage the resources; this will complicate the analysis even more. Finally, the support of the software's execution is an electronic component, also susceptible to being affected by a fault!

As a result, the pertinence of the cause's designation is linked to the means which are mobilized in order to remedy it. Consequently, faults are often called *adjudged* or *hypothesized* causes.

In Appendix D we provide a study relative to the diagnosis of the flight control system fault of the first flight of Ariane 5 rocket. This study highlights the difficulty to specify the causes of a failure. This case is also interesting, as it shows a situation where the two technologies (hardware and software) intervened in the failure occurrence.

3.2.2 Fault Characterization

Faced with the difficulties evoked in the previous section in identifying faults, we are now going to highlight some significant criteria to characterize the faults. As for failures, it is useful to define a set of criteria which permits the classification of faults. Such a classification allows the proposal of generic means of fault handling adapted to each class, instead of researching specific means for each particular fault.

We will analyze faults according to two main viewpoints: their origin and their nature. Each of them is defined by several criteria.

The *origin* of faults characterizes:

- where is located this fault, that is which is the affected object (the product, the user, their environment)?
- when was the fault introduced into the life cycle (during the product creation or during the operation)?
- who is the author of the fault?

The *nature* of faults identifies:

- its *type*: functional / conceptual faults opposed to technological faults,
- its *intention*: *accidental* faults opposed to *intentional* faults,
- its *duration*: temporary faults opposed to persistent / permanent faults.

These fault characteristics are illustrated in *Figure 3.4*. We thus define a space with two viewpoints and several criteria that will be analyzed in the following sections.

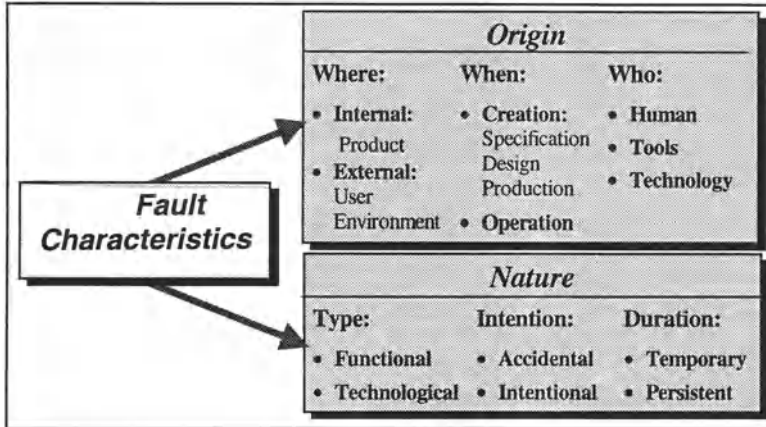


Figure 3.4. Fault characterization

3.2.3 Fault Origin

The origin of a fault is characterized by three criteria studied in the following sub-sections:

- The object at the origin of the fault, by asking *where?*
- The step of the life cycle at which it appears, by asking *when?*
- The entity responsible of its occurrence, by asking *who?*

3.2.3.1 The Infected Object: Where

Relatively to the product's point of view, we consider two classes of faults (see *Figure 3.5*): *internal* and *external faults*.

Internal faults affect the product and are introduced during its life cycle: specification, design, production and use. For example, a software design error in an airfighter control system provoked the turn over of the plane during its first flight to the equator.

An internal fault is for instance an erroneous statement in a program or a short-circuit in an electronic component. If their effects in the form of failures always occur during the operation phase, their causes could originate from any stage of the life cycle. This concerns the design step in the first quoted example (an erroneous statement), whereas the short-circuit in the second example could have been introduced by the manufacturing.

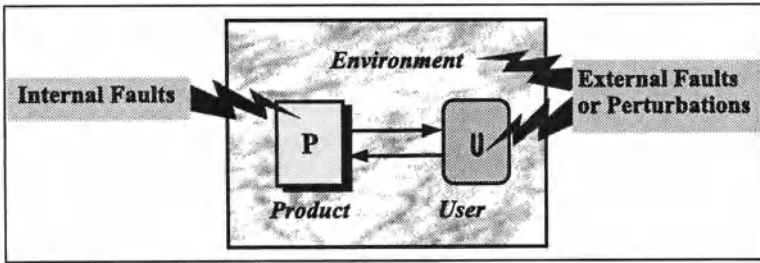


Figure 3.5. Internal and external faults

External faults affect the user or the non-functional environment:

- the *user*: e.g. a system controlling a flexible manufacturing workshop is blocked whilst waiting for a drilling machine (belonging to the process) which will never fulfill its task because of a rupture of a drill, an input value out of range, or a sequence of events not in accordance with the expected scenario;
- the *non-functional environment*: e.g. the coffee distributor provides cups filled only with powder because the water supply has been cut off, an integrated circuit does not work due to an excessive temperature.

The external faults affect the expectations defined by the specifications. They are often called *perturbations* or *aggressions* or else *disturbances*.

We should insist on the fact that in all failure cases, the product does not deliver the expected service, but the object at the origin of this failure is the product itself (internal fault) or not (external fault). Consider as a last example a ‘heavy ion’ bombarding a satellite whose functioning is altered. The origin belongs to the environment whereas it is clearly the satellite which is affected.

3.2.3.2 Occurrence Phase: When

Even if failures only occur during the operational phase, faults which provoke these failures arise during the diverse phases of the product’s life cycle: specification, design, production and operation.

During the specification. It might seem strange to talk of faults arising during this stage as the specifications constitute the reference document which defines the expected services (cf. *Figure 3.2*). However, some problems are actually possible. For example, an incomplete definition of the service, which has to be delivered by the product, leads to different interpretations by the client, the designer and the user. The user notices failures in the service provided (the service understood by the designer) as it is different to the one expected (the service understood by the user).

During the design. All designers have encountered these faults. They can arise during each one of the design steps, from the architectural definition until the final implementation: at behavioral, structural and technological levels.

During the production. An example of fault affecting software technology is about a change in the characteristics of an execution environment (a new hardware processor or operating system) whose performances are no longer sufficient to respect the deadlines of tasks of a real-time application. When hardware technology is concerned, a failure can be due to a short-circuit or to a rupture of a wire during the production phase of a PCB (bad insertion, bad soldering).

During the operation. Such faults can come from an elevation of the environment's temperature which provokes modifications of the electronic equipment's capability.

Note. Faults appearing during the first three stages are also known as *creation faults*.

3.2.3.3 Entity Responsible: Who

Faults result from:

- the human activity which intervenes in the transformational process (specification, design, production) used to create the product: erroneous choice of methods and technologies, bad interpretation of specifications, wrong application of chosen methods, etc.;
- the automated *tools* used during this process: automatic design of a logical circuit, compilation of a program, insertion of electronic components by a production machine, etc.;
- the product's technology: a weak resistance to aggressions of a component (this resistance depends on the hardware technology used), the use of a programming language which has an imprecise semantic may lead to different interpretations by the user and the compiler's creator;
- the user of the product (a human user, an industrial process), associated with environmental constraints.

3.2.4 Nature of the Fault

We will successively consider the *type* (*functional, technological*), the *intention* (*accidental, intentional*), and the *duration* (*permanent, temporary*).

3.2.4.1 Type: Functional and Technological Faults

A second parameter permits the classification to be refined. Whatever their origin is, the faults can be separated into two categories:

- The *functional*, or *conceptual faults* (also called *human-made faults*), which affect the way a product is specified, designed, produced or used. An incorrect design implied by an omission of one piece of specification is an example of functional fault.
- The *technological faults* (also called *physical faults*, or *hardware faults* in the case of electronic components), which affect the implementation means during the production and/or utilization. A cut in a wire linking two components is an example of technological fault.

Functional faults

Functional faults concern the intrinsic functionality of a product. They can be present when the product is supplied (for example a design fault) or could be due to incorrect use of this product (an operational fault). Functional faults have a precise cause, upon which one could have acted at a given moment in order to avoid them. This group contains faults coming from bad interpretation and/or transformation made during the life cycle:

- *specification and design faults* due to humans (the partners of the initial contract and the persons in charge of the design) and to the means used to model or to transform the models,
- *production faults* due to humans and technical means involved (essentially the manufacturing equipment),
- *operational faults* due to the functional environment of the product (human users, process, etc.).

Failures due to functional faults are named *systemic failures*.

Technological faults

Technological faults affect the implementation means. They can result from random or temporal problems, such as a transient problem of the machine manufacturing a component, or a physical defect occurring in this component due to an ageing problem or an external aggression.

This group concerns the physical components (electronic or mechanical) employed in the manufacturing of the product. This could imply hardware breakdowns occurring at any moment during the production and/or operation by affecting a product which functions correctly: for example, a transistor which is suddenly blocked in a 'non-conducting' state, or an electrical line

which is influenced by an electromagnetic perturbation. The probability of occurrence of such events depends on the chosen technology, the production techniques, as well as the non-functional environment of the product (temperature, mechanical shocks and vibrations, etc.). These faults are assessed by statistical studies on *reliability*. We should note however that the design and production have an influence on the final product's reliability due to the choice of technology, the electronic structure of transistors and final mounting and assembly techniques.

We often speak of *hardware faults* and *physical faults* where *technological faults* are concerned, as they essentially affect the hardware technology (electronics in our case). In effect, there is no real software ageing phenomenon, and the potential problems due to the manufacturing are negligible. However, phenomena similar to those of hardware technology nowadays affect more and more software applications. Such a situation is illustrated by a program whose production necessitates the use of non-adapted components, or whose behavior varies during the course of time. This is the case of applications supported by an operating system whose version changes as the producer modifies slightly the characteristics (e.g. temporal), leading finally to modification of the delivered service.

Failures due to technological faults are named *disruptive failures* or *disruptions*.

The perturbations due to the user (a human or a machine) induce mainly functional faults (bad use), whereas the perturbations due to the non-functional environment induce mainly technological faults (such as in the case of a raised temperature which could lead to a breakdown of an electronic component). Once more, the ultimate reason for a failure is often very difficult to establish: normal ageing of the component or excessive temperature aggression?

3.2.4.2 Intention

Faults can also be classified according to their *accidental* or on the contrary *intentional* character.

Accidental faults are the most frequent and the ones which will be considered in this book. For example, they can stem from a bad understanding of a document's information during the development phase, or from a bad analysis by the designer which led to an erroneous solution or incorrect use of the technology means at his/her disposal. For example, an engineer has incorrectly used the programming language statements because he/she has not understood the semantic properly. This class of faults also includes keypressing faults.

The *intentional faults* are due to voluntary human aggressions, such as intrusion, sabotage or piracy. They lead to the modification of a system's structure or of a product's behavior. Today, there are numerous examples regarding computing networks.

The border between these two classes can be vague. For example, some technologies are reputed to be dangerous and their use increases the number of faults. This is the case, for example, with a programming language such as C which:

- does not favor a *programming style* which renders the program readable,
- disposes of few fault detection means during compilation,
- proposes features, such as the `goto` statement, which makes checks difficult.

As these features are known, designers should not use such technology which leads to an increase in the number of faults produced. Therefore, it is difficult to say whether such faults are accidental or intentional!

3.2.4.3 Duration

Like failures, faults have temporal attributes leading to two classes:

- *permanent faults*, also called *static faults*, for example, a power supply breakdown which makes an equipment unusable,
- *temporary faults*, also called *dynamic faults* for example, a bad electric contact which depends on the product's position, or a temporary saturation of a computing network.

Temporary faults are divided into two sub-groups, according to their origin.

- *Transient faults* have external causes. For instance, a too numerous number of pieces of data are sent to the system during a short period.
- *Intermittent faults* are due to internal causes. For instance, a parasitic signal emitted by a part of an electronic system disturbs another part during the operation.

3.3 FAULTS OCCURRING IN THE LIFE CYCLE

In this section, we consider the faults occurring during the various stages of the product life cycle: specification and design faults, production faults, and operational faults. This presentation shows the diversity of the types and origins of faults.

3.3.1 Specification and Design Faults

Faults introduced during the specification and design stages are various, and the reason for their existence is diverse and often difficult to identify. One commits faults by ignorance, by negligence or omission, by incompetence, by misfortune, and even voluntarily in some cases (an aspect which is not considered here). Nonetheless, the origins of these functional faults can be classified into three groups:

- initial faults arising from incomplete or incorrect specifications,
- faults arising from the top-down design process,
- faults arising from non-functional constraints.

These three groups are not totally independent. Faults can therefore belong to several groups corresponding to complex situations. This decomposition, refined in the following paragraphs, simplifies our study.

3.3.1.1 Specifications

Stemming from the contract generally written in natural language and established between the client, the designer and the user, the specifications are therefore rarely formal. A lot of *incompleteness* and *inconsistency* cases remain. They are at the origin of numerous faults. After their detection, it is necessary to precise missing information or to modify bad elements, in order to improve the specification.

Incompleteness characterizes a product's definition that can lead to several interpretations. The 'non specification' of a system's behavior for an input data or a sequence of input values is an example of incompleteness. Incompleteness is therefore associated with the semantic of specification elements. An incompleteness situation leads to a fault if the missing piece of information is useful and generates a bad interpretation. Otherwise, incompleteness avoids the expression of non-useful information or gives some degrees of freedom to the designer. For instance, the output value associated with a given input value can be absent if the user never applies this input value. When a piece of information is missing, the designer is free to interpret this 'non specification' in the best interests of the product (cost optimization, execution speed, etc.). On the other hand, if the missing element comes from a mistake, the final product can behave incorrectly.

Due to a lack in the specifications, the system obtained at the end of design can have a functioning greater or equal to that of its specifications. For example, an electronics circuit accepts all input data (as long as the input sequence stays in a functioning mode which guarantees sufficient time between two successive input values to produce an output), even if these

values have not been defined in the specifications; hence, it provides an output value for each input value

These incomplete specifications can, on the contrary, lead to a dysfunction if the actual operation does not support use outside of the planned domains. For example, in the case of software, the supplying of an input value non-planned by the designer, due to a restrictive interpretation of too vague specifications, can lead to the following situation:

- a persistent failure if the program execution stops,
- a dynamic and temporary failure due to a random transient behavior if the program is re-initialized after the failure.

The *inconsistency* characteristic corresponds to another variety of problems. An *inconsistency* expresses a contradiction between several definitions or properties of one or several elements of the specifications. An example is the definition of two different behaviors for the same input applied to a system being in the same state. The technology implementation will perhaps solve this conflict:

- Whether by giving advantage to one of these contradictory behaviors; for example, if we simultaneously act on the 'Set' and the 'Clear' (or Reset) inputs of a flip-flop, it is the 'Clear' which is dominant.
- Or by creating a third behavior (by combination of the values which lead to a new value); for example, if we simultaneously switch on the red light and the green light of a traffic light system, therefore the two lights switch on simultaneously by OR combination of the two light control vectors (Red, Orange, Green): $(100) \text{ OR } (001) = (101)$.

3.3.1.2 Functional Design

At design time, a system is modeled by expressing a structural composition of *components* also called *modules* or *sub-systems*. Faults introduced in the design phase belong to two classes: *component faults* or *interaction faults*.

Component faults (or **module faults**). A component fault occurs if a functional component does not fulfill the mission which has been defined during its specification; for instance, a floating point multiplier or a calculating subprogram gives an erroneous result for a particular input configuration or on the contrary in a systematic way.

Interaction faults. The components are correctly designed, but their interactions can cause problems:

- the interface specifications are erroneously taken into account: e.g. there is an incompatibility between data formats, or the constraints on the order

of the subprogram calls have not been considered (for instance, the subprogram `init` of a package must be called before other subprograms offered by this package),

- the inter-relations are erroneously implemented: for example, the design specifies an exchange protocol between two tasks which is not correctly realized.

The interaction faults become increasingly preponderant due to the fact that a large part of design today consists in assembling acquired components (COTS: Components On The Shelf). The faults are due to an erroneous assembling or to a correct assembling of the components whose effective functionality has been badly understood by the designer. During a development, the complexity of the designed system and the diversity of technologies used often necessitate the separation of the work into teams distributed into several companies. Numerous integration faults, that is interaction faults, therefore occur.

3.3.1.3 Technological Constraints

Often unknown or badly formalized, the *technological constraints* imposed on a product's development are sources of numerous faults. We introduce in the following sub-sections two types of constraints: *technical constraints* and *reusability constraints*.

General technical constraints

General technical constraints are often included in the design requirements. They precise:

- certain technological choices of components: for example CMOS components for the electronics implementation, or Ada language for the software programming,
- design means, e.g. the use of UML design model,
- constraints on the size of a product (volume of software code, surface of the integrated circuit),
- constraints on the electrical consumption (important constraint for isolated or embedded systems),
- assembly constraints, cost constraints, etc.

Moreover, other constraints, non-necessarily expressed during the specifications, appear during the design phase. They are constraints on the available resources. For instance, the program task number is limited by the executive software, the number of units which can be addressed on a Bus

depends on the Bus characteristics, the size of available main memory, the number of authorized interrupts are constrained by the hardware platform, and so on.

If these constraints are not well known, or if they are not correctly taken into account at design time, then a faulty system is produced. For instance, a memory overflow is raised at program run-time when the memory top is reached. This example shows again how fault location is difficult: is the failure due to a too small memory size or a too large program memory allocation?

Reusability

Whether in hardware or software domain, the designs are lengthy and expensive. It is therefore tempting and interesting to *reuse* the components already designed for other projects by adapting them to a new context: a circuit or a calculation subprogram, a special register or counter, a FIFO memory, a BUS coupler, etc.

This reuse is sometimes obligatory. For example, a software application runs under a given operating system. This application reuses the functionality offered by this system such as the input/output primitives. This obligation to reuse arises also from the constraints of portability. For example, the embedded programs do not generally access directly to the hardware resources of the electronic board which supports them. These programs call the primitives of a BSP (Board Support Package) which provides an abstract view of the hardware. For example, the applicative program makes use of a primitive function to write on a port without the knowledge of the physical port address. Hence, the software is more portable as the hardware may be changed as long as it uses a BSP offering identical functions. In this way, the set hardware plus BSP constitute a reused module.

Certain functions of the reused module can be of no use in the new context or can be used in a particular restrictive way: they are therefore *redundant* (we will discuss this word later). The redundancy resulting from the reuse of modules is frequent. For example, in order to carry out an addition operation in a circuit we reuse an arithmetic addition/subtraction unit: in this context, the redundant subtraction function will not be used.

Reuse is frequently employed because it is convenient and it is supposed to reduce costs and faults. This argument is globally true, but it should be emphasized that reutilization can typically lead to new sources of problems. On the one hand, the insertion of a component whose subtleties and weaknesses are not known leads to interaction faults; on the other hand, the induced redundancy creates large verification problems, which we will study afterwards. This is why the modules inserted have to be perfectly defined and possess standard interfaces.

In order to avoid these problems, we could perhaps a posteriori think to get rid of all parts that are functionally or structurally redundant and therefore useless to a system's mission. In reality, this suppression is not desired, as it is likely to introduce new faults.

3.3.2 Production Faults

We will analyze separately the two hardware and software domains, as they present problems which are completely different during production.

3.3.2.1 Hardware Technology

Integrated circuits

We consider first of all the particular and very significant issues of the manufacturing of an integrated MOS circuit. The design stage provides a geometric model of different masks used in manufacturing. Roughly speaking, and in order to simplify, each technological layer has a design associated with it: N and P diffusions define the transistor channels and some conduction lines, the polysilicon level defines the gate electrodes of the transistors and some conduction lines, the metal level defines the interconnections between transistor structures and the links with the input and output pads. The manufacturing is going to interpret this information so that it can structure a slice of pure silicon wafer (disk of silicon of some 15cm in diameter and less than 1mm thick) in the form of an array of identical chips, each chip making the desired product. The process which transforms the silicon wafer into an electronic structure is long and calls for very sophisticated specialized technological equipment (ionic implanters, e-beams, diffusion ovens, epitaxy machines, deposition and etching machines and many more), and it has to function in extremely severe conditions (temperature, duration, dusts, etc.).

After the wafer processing, the resulting wafer is cut into dies which are then mounted in their final plastic or ceramic packages (dual-in-line, flat pack, surface mounted, etc.). This implies mechanical and soldering operations before obtaining the final chips that will be put on the market.

As previously mentioned, faults come from the use of manufacturing equipment. The information provided by the design of masks has of course got to be compatible with the equipment used during manufacturing. A machine's control file has to be understood correctly: compatibility of description formats or correct initialization of machines, etc.

This sophisticated equipment needs precise and frequent settings (treatment duration, temperature, flow, intensity, position), as do the gauging

operations. The quality of the physico-chemical hardware used (crystalline structure, fluids, etc.) also conditions the quality of the circuit produced.

Moreover, this process has to take place with an extremely strict control of the environment: temperature, hygrometry degree, elimination of all dust or particles which could provoke flaws (hence, different classes of dust removing techniques of white rooms have been defined). Thus, the dust can create flaws by optical or chemical interference. If the complexity of integrated circuits is meant to double every 18 months (according to Moore's empirical law), this then implies that the manufacturing costs will double every 4 years!

As a consequence of this complexity, numerous faults can be introduced during the production of integrated circuits.

Electronic board systems

The manufacturing of an electronic product involves a succession of assembly stages and the integration of components and equipment. Each of these stages is an occasion for faults to appear, and this occurs despite the use of specialized equipment and qualified personnel. For example, electronic components are inserted on printed circuit boards (PCB) by automatic insertion machines, eventually aided by an operator. Insertion faults can therefore be produced: incorrect mounting, a pin folded, an incorrect electrical contact. This is also the case with incorrectly soldered pins by the component's welding machine. Boards are linked together thanks to diverse and varied connectors defined by different standards. Thus, the connector industry causes numerous problems due to bad quality contacts (mechanical problems, oxidation problems, etc.).

3.3.2.2 Software Technology

We consider that design ends by the providing of a program written in a language (Ada, C, etc.). The production therefore consists in obtaining an executable program for the execution machine and then by duplicating it onto physical medium (magnetic disks, CD-ROM, ROM, EEPROM, etc.). Naturally, these two phases are both sources of faults.

The executable code is generally produced automatically from a source program by a compiler. The code generated has to be semantically equivalent to the code source. That is, its execution by the target computer has to produce a behavior equivalent to the one obtained by the interpretation of the source program using the language semantic (described in the reference manual of this language.) Two major causes thwart this result:

- failure of the compiler,

- hazards in the programming language's semantic.

Compilers are not always safe tools, in particular when the language or the compiler is recent. For these reasons, a lot of firms prefer to use an 'old' language and a compiler whose bugs have been fixed or are well known, than to access to up-to-date technological means.

Hazards are associated with the semantics of the programming languages. The significance or interpretation of their features can present uncertainties. We have seen that in the design phase, these lacks of precision can generate failures of the system designed because the client and the designer of the system (or sub-system) can have two different interpretations, although in agreement with the imprecise specifications. The same situation exists for language users (e.g. program designers) and the people carrying out the compiler: they can give two different interpretations to imprecise features of a programming language. Incompleteness situations do not however imply flaws in the language; sometimes they are on the contrary indispensable. For example, the execution duration of a statement such as ' $I := J + K ;$ ' is not defined by the standard of a programming language, whereas the actual duration of execution will have consequences on the application's performance and could lead to a failure if the performance is too weak (such as in real-time applications). Now, to fix a standard in a language regarding the execution time of each statement would be stupid, as this would not allow it to benefit from the permanent improvement of processor speed. Although indispensable, the imprecise factors of the semantics of programming languages can therefore lead to failures.

The duplication of executable files to market the software onto diskettes or optical disks is another source of faults, as the physical media of recording either magnetic, optical or other, inevitably provoke flaws: parasitic signals during the recording or transmission, flaws by punctual alteration of the media and other problems. This class of faults will be fairly easy to manage using redundant coding techniques intensively employed by all information supports, magnetic tapes, magnetic or magneto-optical disks, audio or digital CD ROM, electrical or Hertzian transmission means.

3.3.3 Operational Faults

We are now at the ultimate stage of the product's life cycle, and new problems arise! During operation, technological faults appear on hardware devices, and perturbations are provoked by the non-functional environment and the user (process and/or human operator). They are called *operational faults*.

3.3.3.1 Technological Faults

Hardware faults affect the product during its active life. They are linked to the used technology and assembly techniques, but also to the conditions of the environment. Reliability permits us to predict in a statistical manner whether a population of products has the capacity to survive. This notion will be discussed in detail in Chapter 7. However, we do not know what the next breakdown will be and when it will appear in our product.

The occurrence of faults can be considered as a probabilistic process, function of time. In general, we make the hypothesis that the interactions between the product and the user do not have an influence on the probability laws. However, this is not always true: for instance, a light bulb has a higher probability of breaking down when it is 'switched on' than when it is 'on', 'off', or 'switched off'. Knowledge of reliability allows us to anticipate failures, but not to prevent them during the active life.

Non-functional environment has an influence on the technological fault occurrence. For example, if the temperature increases, the reliability of a circuit is degraded. If the environment's parameters do not correspond to those specified for the product, faults could then occur as a result. Thus, if the utilization temperature exceeds the norm associated with the product (for example, maximal temperature = $+70^{\circ}\text{C}$), this product cannot then be used due to a strong degradation of its reliability. Another example is that of the standards of space radiation for products embedded in satellites. In particular, magnetic media cannot be used in an efficient manner without costly and drastic protection.

3.3.3.2 Faults Caused by the Functional Environment

Functional faults are caused by the user whose behavior is not conforming to that planned by the specifications. For example, if an electronic thermometer has been designed to display temperatures with 2 decimal figures, it will not function correctly if it receives from sensor values superior or equal to 100°C !

Another type of functional faults is due to the human user. Consider for example a product whose characteristics and user instructions are described by a user manual. The user may commit faults. For example, a video recorder whose channels have not been set cannot record a TV program. The user should have first performed a channel setting.

The integration of a product into its final context introduces different problems. For example, a program correctly designed and produced is badly adapted to the application's environment (the operating system incorrectly manages this program). Another example is an automatic control system

which functions badly because it does not receive a correct initialization from its environment (resetting the internal state and some variables).

The specification may be correct but the physical insertion of the product incorrect: for example, a connector is incorrectly plugged or the selected connection port is not the right one, and so on. This is the case of a video recorder badly connected to the television: the user cannot watch the film, or the sound and image will be of bad quality.

3.4 EXAMPLES OF FUNCTIONAL FAULTS ALTERING A DRINKS DISTRIBUTOR

This section deals with the study of the drinks distributors already discussed in Chapter 2. It shows some examples of specification and design faults as well as their consequences. The specifications have been slightly modified in order to reveal some interesting faults.

3.4.1 Description of the Product

Consider the distributor represented in *Figure 3.6*, which has to provide hot drinks such as coffee, tea or chocolate. The machine has a slot to put coins, three selection buttons for the drinks (1: coffee, 2: tea, 3: chocolate), a cancel button, a place where the change is delivered and/or the money is returned, and a place where the selected drink is delivered.

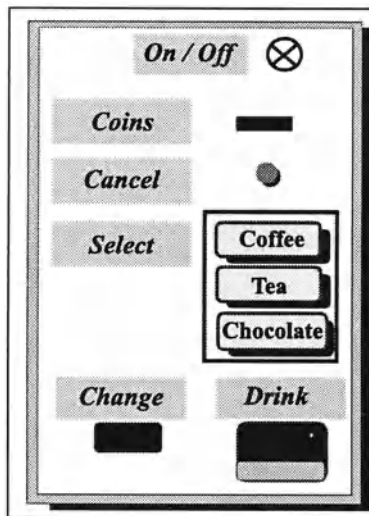


Figure 3.6. Drinks distributor

The distributor behavior specifies a cyclic treatment. One should first insert the money, then choose a drink, recuperate the change and finally take the drink. If we press on the cancel button, the coins inserted are given back and the cycle is cancelled. Then, a new cycle can be processed.

3.4.2 Faults Due to Functional Specifications

These first specifications are sources of problems because they are *incomplete*. Indeed, as no deadline is defined for cancellation, an interpretation of this specification could lead to a machine giving back coins inserted when the drink has already been distributed, if the user presses on the cancel button after this distribution.

What will happen if the user inserts new coins before the cycle has finished? What will happen if the user changes his/her selection? The specifications do not give any indication. Therefore, several interpretations are possible, leading to different products and uses. In particular, if the distributor user and manufacturer make divergent interpretations, a failure will occur at operation time.

These specifications are therefore insufficient. We complete them, saying that cancellation cannot be taken into account if the choice of a drink has been made (or validated by a special 'validation button' which should be added). We add a red/green light that indicates that the machine is being handling a delivery. When the light is green (indicating that the machine is ready), the cycle starts by the insertion of money; this light then goes red until the end of its current cycle. We specify that the first selected beverage is the only one to be taken into account, or we add a validation button.

3.4.3 Faults Due to Technological Constraints

We finally examine the influence of certain resource constraints. This distributor effectively manages resources: change, cups, spoons, sugar, coffee, tea, chocolate and water. The previous specifications, even consistent and complete, do not take into account the natural limitations of these resources.

What should be done when the resources run out? To prevent access by switching the red light on? If there is no more coffee, why not allowing access to the other available drinks (tea or chocolate)? In this last case, the machine must authorize the choice to be modified or cancelled with money returned if the desired drink is not available. If all the coins necessary for any change are not available, the machine should be used with the exact money amount: thus the service delivered is extended.

In all these cases, the cancel button allows the user to get his/her money back as long as a drink has not been selected to be delivered. Cancellation is an essential feature of such machine.

Each omission of constraints in the specifications can lead to a failure in the service delivered to the user, leading him/her to be unsatisfied: loss of change, missing cup, incorrect choice which cannot be cancelled, etc.

3.4.4 Design Faults

The design is going to lead to a structure of interconnected modules, such as the one introduced in Chapter 2. Here the design proposes four modules:

- a module managing the cyclic treatment, asking for services provided by the others modules,
- a module getting the coins inserted, computing the sum introduced, and managing the money to give back,
- a module responsible for the drink delivery,
- and a module in charge of the management of the resources and the anomalies.

An example of a module's design fault could lead to a bad calculation of the sum inserted and the money to be given back. The consequences could be to ask too much or too little money, or to give back too much or too little money.

A first example of synchronization flaw between modules could result in a blockage in the cycle: the management module waits forever the end of the work of another module. On the contrary, the passage to the next stage when the current stage is not completed is a second example of failure cause. For instance, the end of the cycle occurs when the service has not yet been carried out, allowing a new client to be served at the same time.

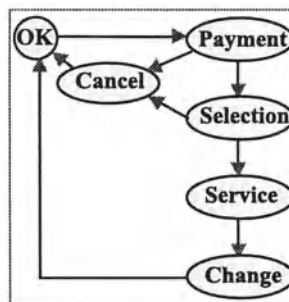


Figure 3.7. Global functional graph of a distributor

As faults are introduced during the various stages of the product life cycle, it is difficult to determine the source of a failure just from a simple external observation. It is necessary to master all the stages, including the initial specifications and their multiple amendments often required by the design. Formal design models bring help to the research of faults, with the aim to eliminate them. The study of a behavioral graph modeling the stages and their transitions can help us to imagine different types of functional faults. *Figure 3.7* shows an example of such a graph. Exercise 3.2 proposes the analysis of this graph.

3.5 INTERESTS AND LIMITS OF FAULT CLASSES

3.5.1 Simplified Classification

As causes of failures, faults are numerous and varied, sometimes predictable but always difficult to identify. They are directly due to human, the tools he/she uses, the ageing phenomena, or aggressions coming from the functional and non-functional environment. Faults are produced during the specification, design, production and operation stages of the life cycle. The effects of these faults as failures are however uniquely perceptible during the product's operation phase. After this broad exploration of the fault classes made in the preceding sections, we are going to simplify the classification given in *Figure 3.4*, in order to facilitate the presentation of the following chapters. Hence, we will put aside the 'who' sub-classes of the 'origin' of faults, and the 'intention' and 'duration' sub-classes of the 'nature' of faults.

These characteristics will be discussed when necessary. Three main criteria remain:

- the *type*: functional and technological faults,
- the *origin*: the product, the user and the non-functional environment,
- the *occurrence stage*: specification, design, production and operation.

A first study consists in establishing if relationships exist between the values taken by these three criteria, that is to say knowing if they are independent or not. *Figure 3.8* summarizes these relationships. Functional faults appear principally during the specification and design stages, and sometimes during production and operation due to external perturbations. Technological faults happen during the production and operation stages; they are influenced by perturbations from the non-functional environment.

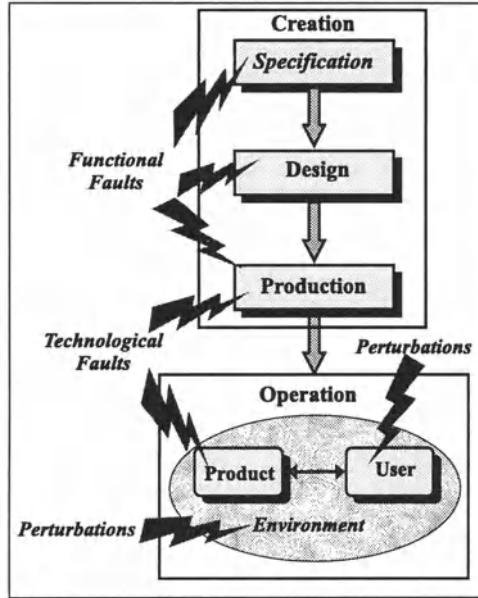


Figure 3.8. Synthesis of failure causes

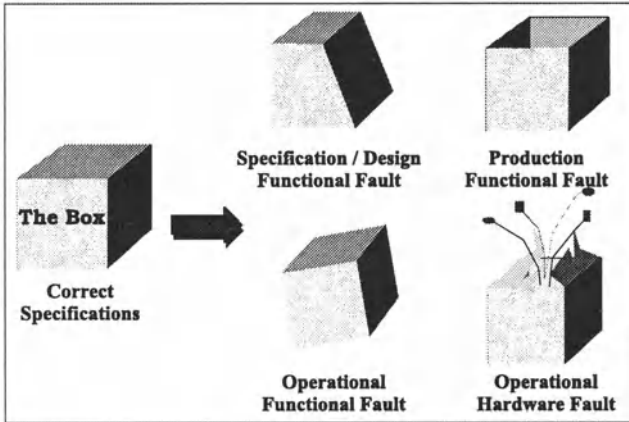


Figure 3.9. Caricature of fault classes

Figure 3.9 shows a caricatured example of problems affecting the life cycle of a 'box': three functional faults arise at specification/design, during the production and use, and a technological fault affects the operation.

The fault classification and the study of the relationships between the classes allow therefore the clarification of problems to be treated according to the stages of the life cycle and the agents at the origin of the faults. We will thus be able to select the means of fault handling appropriate to each

case. The definition of fault classes facilitates the research of generic means, that is to say, which are valuable for all faults belonging to one class. If no fault classes were defined, no general solution would be possible because each solution would be unique to each particular fault. The mastering of faults would therefore be a collection of individual experiences.

3.5.2 Limitations of the Classification

We should note again that it is often difficult to identify the exact cause of a failure. Several faults can lead to identical effects and therefore be equivalent. In the same way, it is not easy to precise the origin of a given fault. For example, the absence of a connection between two logical electronic components could be due to a functional fault during design, or because of a production fault (a forgotten connection), or to a hardware fault (breakdown leading to a rupture of the electrical wire), or even due to an external perturbation, or else to sabotage! The most often, the diagnosis, that is to say the research of the failure cause and/or of the fault origin, is limited by the investigation and the observation means available to the external operator (called the *tester*).

The border between the defined classes is often vague. Even if a fault has been clearly identified, it can be difficult to put it in a given class. For example, a technological fault such as a cut in the connection between two components could be due to the environment (because of a too high temperature), a production problem, or even a bad design choice.

Moreover, faults have a cumulative character which complicates the analysis. Faults occurring during the different stages of the life cycle will persist and, by combining amongst themselves, will create failures during the operation phase. It is only when they reveal themselves as failures that we see their negative and even catastrophic consequences on the application.

The limits which have just been exposed as well as all the particular cases of special malicious faults do not however change the significance of our classification which has an educational interest, and facilitate the organization of means to fight against faults.

3.5.3 Protection Against Faults and their Effects

Even if they correspond to complex problems and phenomena, faults are not a fatality that has to be endured. They have an origin upon which we can act. Indeed, fault appearance is not independent of the methods, techniques and technologies used all along the product's life cycle. Experience shows that some methods, techniques and technology produce fewer faults than others. This knowledge will allow the improvement of the dependability of

products by reducing the probability of the appearance of faults and therefore failures. We will come back to this point in the second and third parts of this book.

Faults depend also on the technologies used to built the product. We consider here computer systems which have hardware parts (e.g. electronic components) and software parts. Now, these two types of logical products do not have the same type of faults: in particular, the hardware ageing fault phenomena which affect electronic components during their functioning with statistical laws do not exist in software parts.

In Chapter 5 we will provide information on the main fault models used for hardware and software technologies.

3.6 EXERCISES

Exercise 3.1. Failures of the distributor

Imagine several failures of the drinks distributor presented in section 4.

1. A static failure.
2. A dynamic failure.
3. A temporary failure.
4. A static and persistent failure.

Exercise 3.2. Faults of the drinks distributor

Go back to the study of a drinks distributor, looking at the global behavioral graph (see *Figure 3.7*).

1. Imagine several types of functional and hardware faults and show how they transform the graph. What failures do they lead to?
2. Which type of faults (and on which part of the graph) affect the user's satisfaction by altering the functioning of the money management (accepting and giving back coins)?
3. Find a functional transformation which allows this distributor to serve several drinks with the same initial amount of money or to give the change back by pressing on *Cancel* (you can add a *Return-Money* button).

Exercise 3.3. Study of a stack

This exercise studies the influence of internal and external faults on failures of a hardware stack. This product, carried out with the aid of a

logical circuit, allows to store data by PUSH operations, and to read them in the opposite order of their recording by POP operations.

A typical failure which can affect the stack consists in an overflow, that is to say executing a PUSH when the stack memory is full. To avoid this problem, we add an output signal called `Stack_Full` which takes '1' when the memory is full and '0' when the stack still has free space. We can nonetheless imagine several faults leading to an overflow despite the presence of this signaling mechanism:

- an internal functional design fault: the size of the stack has been underestimated by the designer,
- an internal hardware fault: a breakdown affects the `Stack_Full` signal and maintains it at '0' value (no signaling) despite an excessive piling up,
- an external fault: the external circuit uses this stack and ignores the signal `Stack_Full`.

We should note that a software implementation of this stack could have been carried out using a package which exports the subprograms PUSH and POP and the exception signal `Stack_Full`. We can imagine similar internal and external functional faults as the preceding ones. A fault equivalent to the technological one of non-transmission of the `Stack_Full` signal will arise if the program language used does not dispose of the exception mechanisms to treat it.

1. For each one of these faults, find a functioning sequence which provokes a stack failure. Does one functioning sequence exists which reveals the presence of one of two different faults (producing the same failure)?
2. Imagine several failure situations that would require the use of a `Stack_Empty` signal.

Exercise 3.4. Study of a program

We consider a program which declares two global variables, *A* and *B*, of Integer type and the two following functions:

```
function F1 return integer is
begin
  A :=A+1;
  return A;
end F1;
function F2 return integer is
begin
  A :=2*A;
  return A;
```


end F1;

Determine the value of B after the execution of the following statement, assuming the initial value $A = 1$:

$B := F1 + F2;$

Chapter 4

Faults and their Effects

In the previous chapter, *faults* have been identified as the generic sources of *failures* which can modify the operation of any given hardware and/or software *product*. In this chapter we will continue the analysis of dependability impairments by specifying the fault notion and introducing the *degradation mechanisms*. These mechanisms gradually transform a *fault* into one or several *errors* (internal effects), then into *failures* which finally have *consequences* on the functional environment and thus deteriorate the mission entrusted to the product (external effects).

The internal and external effects of faults are examined in sections 4.1 and 4.2. Section 4.3 synthesizes the degradation phenomena considered here.

4.1 INTERNAL EFFECTS

4.1.1 Fault

In Chapter 3, faults were defined as *adjudged or hypothesized causes of failures*. The real causes of failures are often difficult to determine and express. Their precise localization and identification depend on the investigation means used to analyze the faulty product. When these investigation means allow the system structure to be examined, faults can often be specified as structure alterations. In this case, a fault can be more precisely defined as follows:

A ***structural fault*** is a non-adequate alteration of the structure of a system.

Such a specific structural definition of faults is considered when dealing with general-purpose design models and more specific gate or transistor hardware models and program models. For example, consider the following program extract:

```
if (A>B)  then ...
           else ...
endif;
```

Let us suppose that the programmer has written 'A>B' instead of 'A>=B'. This is obviously a fault, as the program structure is inadequate.

In the case of electronic circuits, a fault occurs for example if the designer has forgotten a connection between two components, or has used a NAND gate instead of a NOR gate.

One fundamental property of faults in both hardware and software technologies is that they are generally not identifiable as such. A simple observation of the structure of the product cannot help the observer to decide if a fault is present or not. In the first example, the condition 'A>B' is syntactically correct. In the second example, the presence of a NAND gate may be adequate or not. Hence, to judge that a structural element of the system is a fault, other pieces of information are necessary to justify this hypothesis. In fact, to definitely say that a fault exists, the notion of "non adequate alteration" must be explained, specifying what is adequate or not. Sometimes, a reference model provides properties on the adequate or acceptable structures. The syntax of a programming language defines such properties by means of grammar rules; for instance, the omission of the character ')' in the first line of the previous program extract is a fault:

```
if (A>B  then ...
```

Moreover, the elements of a structure are frequently considered as faulty, not in an absolute way but because they produce inadequate or undesirable effects in operation.

The preceding examples illustrate *permanent* faults affecting the structure of the product. In electronics, many other faults can be found which are *temporary* like an electrical interference creating a pulse on a wire. Temporary faults are much more difficult to identify than permanent ones. Some transient or intermittent faults can be modeled as temporal structural modifications of the electronic product. For instance, a parasitic induction between two electrical wires can be considered as a temporary shortcut between these wires.

The fault occurrence is generally invisible from the outside of the product. This fault has no immediate effect on the delivered service of the product. For example, the cut of a wire of a Bus connecting several electronic components does not modify the functioning of the whole product,

as the altered element is not presently used. Unfortunately, during the operational life of the product, this fault will probably provoke a cascade of events leading to one or several failures. We distinguish three main phases in the life of a fault (*Figure 4.1*).

- The fault is *dormant* or *passive*, i.e. it is present in the product but the functioning inside the product is not disturbed. For example, an electronic component is not used although a fault occurred inside it, or an incorrect programming statement has not yet been used at run-time.
- The fault is *active*, i.e. it has an effect on the product functioning. This effect will be defined in the next section as an *error* in an internal component or module. For example, a fault causes an output signal of an electronic component of the product to be wrong, or a programming fault produces the assignment of a bad value in a program's variable or an erroneous branching.
- The error is *propagated* inside the product till it reaches the outputs of the product, hence creating a *failure*.

The transformation of a fault into an error is called the *fault activation*. The mechanism which creates several errors in the product till provoking a failure will be called the *error propagation* mechanism.

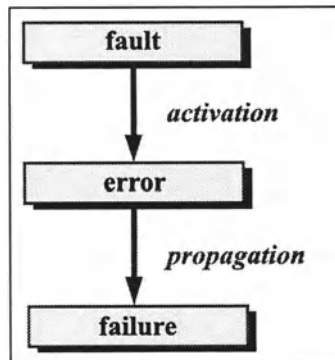


Figure 4.1. Internal effects

4.1.2 Error

As explored in section 2.3 of Chapter 2, the modeling of the behavior of each module of a product structure makes use of *attributes* which characterize this behavior. These attributes can be the external input/output variables and the internal variables memorizing data and control events of the module processing. At run-time, various values are assigned to these

attributes. At a given time, the set of values associated with the attributes defines a *state* of this module. The actual functioning of the module is then defined as transitions from state to state. A defective functioning is perceived when some properties on the state value or on the state evolution are violated. For instance, if the state of a program is defined by the value of a variable `Temperature_of_the_Ice`, then the property:

'`Temperature_of_the_Ice <= 0.0`' is expected.

An **error** occurs in a module when its actual *state* deviates from its desired or intended state.

An error is due to a fault. Thus, a fault modifying the structure of the product will change its functioning, and produce an error.

Let us now consider the following small program extract:

```
if (A>B)   then   Temperature_of_the_Ice := F1(A,B) ;
           else   Temperature_of_the_Ice := F2(A,B) ;
endif;
```

We assume again that the programmer has written '`A>B`' instead of '`A>=B`'. Every time *A* and *B* have the same value, the faulty program executes function *F2* instead of *F1*.

Due to the definition of an error, its characterization depends on the selected attributes and the chosen properties defining the desired or intended states. Consequently, several errors can be associated with this fault according to the attributes and properties chosen.

If `Temperature_of_the_Ice` is considered as an attribute of this program, any variation of the processed value from the expected one could create an error. Unfortunately, the expected value is not a priori precisely known as the program was created to calculate them. Therefore, properties must be defined in order to identify the presence of an error, such as '`Temperature_of_the_Ice <= 0.0`'. This property characterizes the acceptable states. Other properties concern state evolutions. Thus, another property could have been: 'the difference between two consecutive values of `Temperature_of_the_Ice` is bounded by 1°C'.

For the same program, one could define another set of attributes, the two parameters *A* and *B* of function *F2*, and choose the following property: 'the specification of *F2* assumes that $A \neq B$ '.

Concerning hardware, let us consider a NAND gate with two inputs, *A* and *B*, and one output *C*. In this case, we have a behavioral reference model expressed by the property: $C = (A \cdot B)'$, where the symbols ' \cdot ' and $'$ ' respectively represent the logic AND and the logic complement. Thus, any fault of the transistor structure of this gate producing a wrong value on *C* is perceived as an error if $C \neq (A \cdot B)'$.

On the contrary, the property ‘Temperature_of_the_Ice <= 0.0’ of the software example is not a complete formal model of the expected behavior of this program. Hence all wrong state values produced by this program cannot be characterized as errors by this property.

So, the notion of error is relative:

- to the knowledge we have about the modular structure of the product and about the selection of the attributes of each module,
- to the properties associated with the attributes which define the internal expected state evolution.

In the same manner as failures, errors have temporal characteristics classified according to two independent criteria:

- **static error** which corresponds to stable undesirable state (e.g. a false signal ‘1’ instead of the correct one ‘0’) or **dynamic error** or **transient error** which provokes transient undesirable states (e.g. a transient oscillation on an electric line),
- **permanent error** which alters the module for a long duration (e.g. the output of a module is ‘stuck-at 0’) or **temporary error** which alters the module for a short duration (i.e. the error presence is limited in time).

For instance, consider a system including a sensor which gets data. We assume that disruptions, such as electrical parasites, may alterate the sampled values. If the system acquires this input value only once (for instance, at initialization time), then a persistent error occurs. If the sensor makes periodic sampling, the error can be present only during one period. In that case, it is a temporary error.

4.1.3 Error Propagation

Once a fault has been activated as an error in one module, degradation mechanisms can propagate this error through the product structure until reaching its output variables, thus producing a failure. The **propagation** (or **error diffusion** or **contamination**) is conducted through one or several **error propagation paths**. This process depends on the initial error, the module which has been first disturbed, the structure of the product and the external input sequences applied to the product since the fault has been activated.

For example, let us examine the structure of *Figure 4.2* infected by a fault located in module M_1 . This fault is activated as the initial error inside this module (error 1). Then, it passes to module M_2 (error 2), i.e. at the level of module M_{12} , and finally reaches M_3 and provokes a failure.

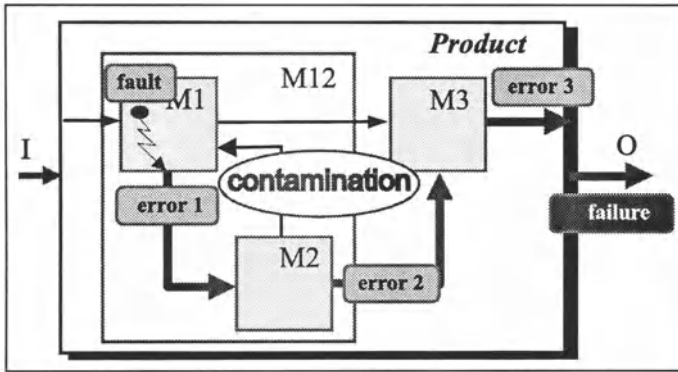


Figure 4.2. Example of error contamination

It should be noted that another description of the product structure by using another modular organization would have modified the sequence of the various errors (error 1 - error 2 - error 3) taking part in the contamination process. In particular, the absence of knowledge about this structure would have suppressed all internal errors! Thus, the propagation mechanism depends on the modular structure of the product and the observation level of the analysis.

A same fault can produce different errors and different failures at different moments of the product's useful life. These effects depend on the fault location and the activity of the product during and after the fault's occurrence. In particular, faults do not necessarily raise errors, and errors do not necessarily imply failures. It depends on the structure of the product and the input sequences which are applied to the product during its use. Let us illustrate this aspect with the program extract example of sub-section 4.1.2. For a given value v of $A=B$, we suppose that $F2(A,B)$ is erroneously executed instead of $F1(A,B)$. If $F2(v,v) = F1(v,v)$, then there will be no error and, hence no possible failure. Hence, in that case the error contamination is stopped.

The qualifiers *static* and *dynamic* are associated with errors and failures. These properties are not necessarily preserved along the propagation paths in the product's structure. Thus, a dynamic error can be propagated as a static error. For example, a register receiving a pulse on the data Bus (dynamic error) can record wrong data (then it becomes a static error).

Moreover, an error created at any moment can evolve with time according to the product activity:

- it can disappear, the operation becoming correct again (this error is also called *overwritten error*),
- on the contrary, it can become worse by a degradation mechanism.

The reasons for these phenomena are to be found in:

- the intrinsic evolution of the initial fault with time, e.g. temporal drift of the response time of electronic components,
- the normal evolution of the product operation, i.e. the activity of the modules evolves with the input sequences coming from the functional environment, hence fault activation and/or error propagation conditions can also be modified.

An error may be temporary, either because the fault is itself temporary, or because the operation of the product activates the faulty module during a short duration. Let us take the example of a parasitic signal in a control circuit: an alpha particle may change the state of a RAM (a word is erroneously changed) and induce a wrong value of a variable storing the value of a sensor used for a control algorithm. This error is temporary because, it will disappear at the next actualization of this variable.

A permanent fault of a computing component (such as a 'stuck-at' electronic fault) may lead to incorrect processing for very special and rarely applied input values. Here also, the error may be temporary.

The degradation of the behavior of a product is due to cumulative effects involving several faults and errors which progressively affect more and more functions of the product. For example, the presence of faults in a communication Bus connecting several processing units can progressively disturb the operation of these units when they are using the Bus.

4.1.4 Latency

A fault remains passive until an error is produced in a module of the structure of the product. We call *initial activation* the first occurrence of an error provoked by the fault (illustrated in *Figure 4.3*). This error is known as *primitive error* or *immediate error*. In the case of the example of *Figure 4.2*, the initial activation causes 'error 1' in module M_1 .

Latency is the meantime between the fault occurrence and its initial activation as an error.

Where software systems are concerned, faults are generally introduced during the development phases. On the contrary, faults can also occur at any time of the operation phase of an electronic product. For the two technologies, errors occur at run-time.

The *latency* value depends on four main parameters:

- The module containing the fault: the latency is high if this module is rarely used during the mission of the product.

- The moment of the apparition of the fault, e.g. the module altered is used only at the initialization of the mission, hence its latency is small during this period and high during the rest of the mission.
- The way the product is used at the occurrence of the fault, e.g. the latency is high because the product does not presently use the infected module.
- The ‘observation’ level given to the product, that is to say the precision of the state definition and the properties associated with these states; for instance, a false signal or a wrong variable value may be considered as an error or not according to their action or not on the state attributes and the ability of the properties to perceive them as wrong.

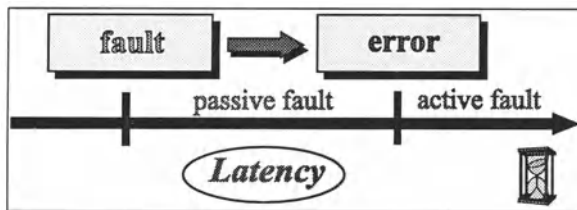


Figure 4.3. Latency

By generalization, *latency* can be extended to the meantime between the occurrence of a fault or an error in a given module and the raising of an error in another given module.

Hence, this latency notion is related to the *observability* notion. Frequently, the latency is referred to as the observation of the fault as a failure at the primary outputs of the product. In this case, the internal states of the modules are not examined and thus no errors may occur.

The assessment of the latency is essentially a statistical process, typically obtained thanks to measurements conducted on samples of the product under investigation. However, it can also be estimated by reference to other products already developed. For example, in the case of software, knowledge coming from past designs allows the latency of the software under development to be estimated.

Some products have an intensive internal activity with a low latency. It is the case of a sequential circuit which counts the average number of external events arriving at high speed (e.g. a particle counter): this circuit has a low level latency (e.g. 1ms). On the contrary, a fire detector can remain in the same waiting state for years, till the occurrence of a fire. Therefore, the presence of a fault may be never observed. This is the reason why fires are periodically simulated. Another example is an ABS car system, which

avoids the blocking of the wheels. Its latency can vary according to the way the driver uses his/her car.

Latency is related to the destructive mechanisms, which transform faults into errors and propagate errors to the product's outputs. In all cases, it delays the appearance of a failure, i.e. the perturbation of the delivered service. This evolution is symbolized in *Figure 4.4*.

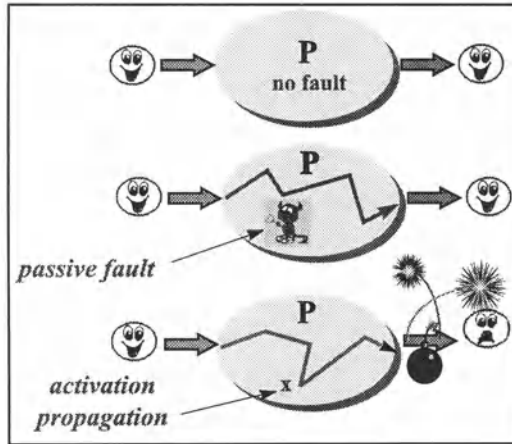


Figure 4.4. Latency caricatures

4.2 EXTERNAL EFFECTS: CONSEQUENCES

4.2.1 External Consequences of Faults

At the end of the contamination mechanism, if propagation has not been stopped, a fault has been transformed into a failure, that is to say a non-desired service delivered by the product in the general framework of its mission. Thus, the product gives incorrect pieces of information (incorrect values, too high response time, spikes, etc.) to its functional environment (the controlled process, the human operator, etc.). We will now analyze the *external consequences* of the failure on the mission, in terms of *seriousness* or *severity* of the perturbations. We identify four main grades of consequences of failures on the mission: *benign*, *significant*, *serious* and *catastrophic*.

- **Benign.** The failure has no serious consequences on the mission which carries on normally. For example, if a text-processor fails, the user can enter his/her text again from the previous back up of the file text (if such back-up file exists). It is also known as *minor* failure.

- **Significant.** The mission is disturbed and the efficiency of the delivered service is reduced; for instance, the failure has economic consequences in terms of costs (fixed or proportional to the immobilization duration). This failure is also called *major failure*.
- **Serious.** The mission is greatly disturbed, the security margins being dangerously reduced; for example, an automated process control has been stopped for a day inducing production loss. This failure is also called *dangerous failure*.
- **Catastrophic or disastrous.** The effects are unacceptable: the mission is stopped with destruction of the product and/or the controlled process (e.g. explosion of a heated distillation balloon) or with human injuries or deaths (e.g. because of the explosion or the emanation of toxic gas).

The severity grade assigned to a given failure is relative. For instance, if the text-processor failure does not allow an industrial project proposal to be completed before the deadline, then this failure is at least significant and not at all benign.

Independently from their seriousness, the consequences of failures can be:

- human (loss of confidence in the product, injury of an operator, etc.),
- economical (for example, a significant consequence is expressed in terms of cost implied by the recovery of the altered function, the diagnosis and the repair of the product and its environment),
- environmental (e.g. air or water pollution).

Three parameters influence the severity of the consequences of a failure:

- the nature of the failure,
- the functional environment which makes use of the product,
- the moment when the failure appeared.

The first parameter concerns the characteristics of the fault which caused the failure, the module which was first altered and the activity which propagated the errors and contaminated the product. One must firstly try to evaluate the loss of functionality of the altered mission. For example, let us consider a regulation system made up of a control unit, an arithmetic processing unit, a memory and an external interface unit. A fault in the control unit can block all the product activity; a fault in the arithmetic unit can, according to the activity, have benign effects (if the application does not make use of it) or severe effects (erroneous computation leading to a dangerous action on the environment).

A same failure of a product has different external consequences according to the functional environment with which the product interacts (second parameter). A hardware fault in a given micro-controller can have quite different effects whether this circuit is used for a game or is integrated in an embedded flight control system.

The precise moment when the fault is revealed as a failure may also be of importance (third parameter). The effects are different whether the product operates in a critical phase (for example, during an aircraft takeoff), or in normal phase (cruise flight), or else is stopped (the aircraft is parked).

During the useful life of the product, the severity of the failure's consequences can evolve and become more critical due to:

- internal reasons (evolution of the errors, appearance of new faults and errors);
- external reasons (modification of the operational conditions, e.g. due to an evolution of the functional environment); for example, a faulty text-processor is firstly used for a non urgent letter (benign consequences), then for an urgent contract proposal (significant or serious);

Let us finally insist on the fact that the seriousness notion is frequently subjective. For instance, let us consider a short breakdown of a TV transmission system embedded in a satellite during an Olympic game program. This failure has minor consequences for people not interested in sport programs. It can be considered as significant by the fans. Finally, the economic cost of this failure can be catastrophic for the program producer because he/she will have to return money paid for the non-delivered advertisements.

Example 4.1. Regulation of a balloon's temperature

To illustrate the notions introduced, let us consider a product intended to control a distiller's temperature. It receives sampled data from sensors and controls the heating gas flow by acting on an electro-valve. Let us suppose a fault in this product which opens the electro-valve at 20% instead of 10%. This fault can arise from a functional design fault (bad specification interpretation, wrong control algorithm, bad hardware design, etc.), a functional or technological manufacturing fault (bad integrated circuit manufacturing, fault during the insertion of a component, etc.), a technological fault occurring during operation (stuck-at fault in a component) or a perturbation coming from the environment of the product (electromagnetic interference). We can imagine several kinds of consequences:

- *significant consequences*: the temperature is incorrect, so the treatment has a smaller yield than the expected one, implying a loss of income,
- *catastrophic consequences*: after a few minutes, the high temperature of the balloon provokes an explosion that destroys the process.

Note that the bad electro-valve control can firstly lead to a decreasing in the yield, and later, by degradation, reach a total loss of the production, obliging to halt the mission.

4.2.2 Inertia of the Functional Environment

In general, the external consequences of a failure do not occur immediately because the functional environment connected to the product presents *inertia* phenomenon (illustrated by *Figure 4.5*). Hence, the explosion of a distillation balloon due to a failure of the heating control system may occur a few minutes after the incorrect electro-valve opening.

The *inertia* is the duration between the occurrence of a failure and the beginning of its external consequences on the mission.

According to the considered application domain, the inertia varies from milliseconds (case of electrical processes) to hours (case of processes from iron & steel industry or civil engineering). Often, for a given failure, the duration between the failure occurrence and the occurrence of its consequences varies. Therefore, inertia is defined as the meantime of the duration values.

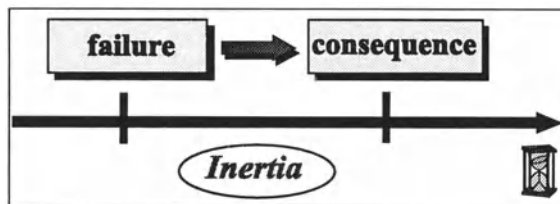


Figure 4.5. Inertia of the functional environment

4.2.3 Completeness and Compatibility

In a first study, the reader may ignore this sub-section as well as sub-section 4.2.4 dealing with *emergence*.

Let us suppose that the initial *contract* between the designer, the client and the user has correctly expressed the specifications of the future product: function, duration, non-functional constraints on the environment and

dependability attributes.

Let us also suppose that no fault has been introduced during the development phases. In that case, the following functional relation is true

$P = \Sigma = S$, where:

- S , Σ and P respectively express the specifications, the functionality of the system resulting from the design phase, and the functionality of the final embedded product,
- and the symbol ‘=’ represents the functional identity (i.e. the provided function is the same as the expected one, taking the operational constraints into account).

The final product used for the mission is now infected by faults that can produce errors and failures. Therefore, P is not identical to S . However, it is useful to separate the three following cases: $P > S$, $P < S$ and $P \neq S$.

- $P > S$: the functionality of the product is greater than the one given by the specifications. The product is able to do more things than given by the specifications: it is said to be *compatible* with them.
- $P < S$: the functionality of the product is smaller than the one of the specifications. The product can do fewer things than given by the specifications: it is a case of implementation *incompleteness*.
- $P \neq S$: the product does not satisfy the specifications. It is a case of *incompatibility*.

The interest of this distinction lies in the interpretation of the failure’s seriousness. One generally accepts the first case as benign, as no failure will be produced. Frequently, the specifications are not formal. They are for example expressed as an English document. So, some situations may be unspecified. It can be deliberate if these situations are known not to occur during the final mission. On the contrary, they may result from omissions in the specifications. Then, the designer will interpret these inaccuracies during the design step. As a result, the product might have more functionality and be able to react to input data that will never be applied by the process.

In the second case, the product may be unable to handle some expected external stimuli (or the reaction may not be predictable), a situation which may be considered as serious.

Finally, the third case is the classical one of failure production already considered.

Note. *Reuse* techniques are increasingly employed in hardware and software design in order to simplify the design process and standardize the production. They can lead to the first case ($P > S$) if some elements of the functionality provided by the reused module are not presently activated.

Unfortunately, the two other situations exist, being at the origin of numerous errors. For instance, a subprogram is reused, as it provides an expected function. However, its designer assumed constraints about the parameters, which are not satisfied in the new utilization.

4.2.4 Influence of the Functional Environment: Emergence

The *external faults* due to the functional environment which interacts with the product are often much more difficult to identify and to master than the *internal faults* which alter the product. Indeed, modeling of the environment is often very complex (mathematically speaking) and implies numerous variables and parameters with much inaccuracy. Moreover, these models are not static: influence of time, temperature, etc. Here, we will not treat faults specific to the functional environment but we will analyze the influence of the interactions between the product and its environment.

The functional relations between the product and its functional environment place constraints on the actual functioning of the product. We call *emergent functionality* the functional part of the product which is really used in the context of the mission. This part is smaller than or equal to the total functionality of the product considered alone without external constraints. We express this property as:

$$Em(P/\Pi) \leq P, \text{ where } \Pi \text{ represents the functional environment.}$$

Embedded in its environment, a well defined and designed product must have a resulting emergent functionality greater or equal to the functionality defined by the specifications. We express this property:

$$Em(P/\Pi) \geq S.$$

This expresses that the real behavior provided by the product in its environment must at least include the specified (i.e. expected) one. So, new sources of functional failures appear, which are connected to the notions of:

- *reuse* of a product already designed and involved in new applications (software functions and packages, integrated circuits), and
- *robustness* of the product faced to perturbations arising from a changing environment.

Reuse. A product P which has been developed for a given known environment Π is embedded in a different environment Π^* . Two situations might occur:

- the functioning of the product is compatible with the new environment, then $Em(P/\Pi^*) \geq S$,

- the functioning of the product is incompatible or incomplete according to the new environment.

The incompatibility comes from faults introduced during the design, production and/or operation stages of the product.

Robustness. The product is supposed to be correctly specified, designed, produced and used. We suppose the occurrence of an external fault, also called perturbation, arising from the environment. This environment changing is noted $\Delta\Pi$ (Π becomes Π^*). We say that *the product accepts this perturbation* if $Em(P/\Pi^*) \geq S$. Then, the product is qualified as robust with regard to this perturbation. In the opposite case, the fault will produce a failure sooner or later.

The **robustness** is the property of a system that defines its capability, when it is aggressed by the environment, to provide a function which is acceptable to the user.

Certain standards define robustness as the characteristic of a product which guarantees that its functionality is maintained even if specified operational and utilization requirements are violated. However, it assumes:

- either a definition of the expected functionality, that is an extension of the specifications of the system,
- or a temporal inactivity of the product which preserves its current state for future normal use.

Our definition does not require a precise expression of what must occur, but just the expression of its acceptance or its rejection, for example using a property. For instance, if a traffic light controller detects a bad use, such as a faulty signal sent by a sensor, the yellow light must wink (specified safe behavior) or the two lights must not be simultaneously green (safe behavior). Furthermore, for this controller the non-reaction and the state preservation may also be adequate to react to temporary faults. The concept of robustness can be linked to the fail-safe property developed in Chapter 17.

4.3 CONCLUSION ON THE EFFECTS OF FAULTS

Figure 4.6 sums up the destructive mechanisms linking faults, errors, failures and external consequences. A fault appears and is activated as an error. The error propagates and contaminates the product until reaching the output; hence, it produces a failure. Finally, the failure has external consequences on the environment.

The vocabulary used in this book was selected from the various hardware and software domains of computer sciences. Unfortunately, dependability

science has not been developed jointly by specialists from these domains. Hence, the keywords have not always the same meaning. In Electronics, the word *failure* often refers to operational values of parameters of a component being out of range. For example, a ‘failure’ of a MOS transistor will disturb the operation of a logical gate. In that context, the *failure* of the MOS component would precede an *error* in the product. Therefore, these notions are recursive. In any case, the notions of fault, error and failure are always relative to the observer’s point of view. Thus, a ‘stuck-at 0’ of a logical output gate may be considered as a fault of the circuit using this gate, an error of the gate module in the complete circuit, or a failure of the gate component. It depends on the investigation level considered.

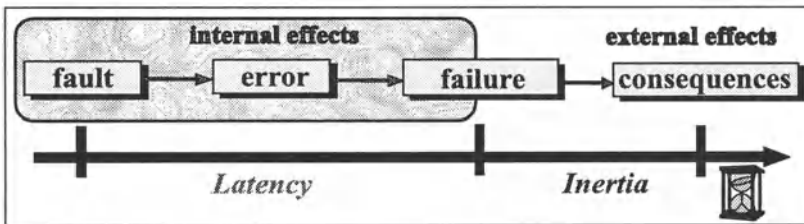


Figure 4.6. Internal and external effects of faults

Latency and inertia are phenomena which delay fault effects: latency delays the internal transformation of faults, and inertia delays the external effects of failures. So, these two phenomena seem to have always positive effects on a mission’s dependability, because they delay the issues and give more time to the *protective mechanisms*, hence avoiding for example a catastrophe. This opinion is really true for inertia.

Concerning the latency, the property is unfortunately not always true. As a matter of fact, latency may have negative effects on the product’s dependability because of the lack of *observability* about faults inside the product. This means that a fault may exist which eventually produces errors that cannot be observed from the outside of the product. In some cases, a high latency may produce an accumulation of invisible errors which, later, may inhibit the protective mechanisms. Such mechanisms are, for example, off-line and on-line testing methods used during the maintenance periods or fault-tolerant techniques embedded in the product to avoid failures. For example, consider a product containing a fault-tolerant mechanism used to handle exactly one error. Thus, if such an error exists at the end of the development process, the presence of the associated fault is masked. For instance, it is not transformed into a failure during a testing operation. However, a second fault occurring and/or activated during the operation will not be tolerated.

Figure 4.7 summarizes the positive and negative effects of latency and inertia. Latency and inertia measurements are fundamentally statistical information difficult to assess with accuracy. Therefore, the practical interest of their analysis is more qualitative than quantitative.

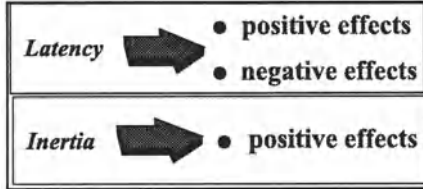


Figure 4.7. Positive and negative effects of latency and inertia

Example 4.2. Analogy with the human pathology

The observation of living beings gives an endless source of examples to illustrate and enlighten the fundamental notions implied in the destructive and the protective mechanisms we are studying here. Faced with the relentless degradation mechanisms, life has cleverly developed a wide range of protective mechanisms to detect and correct the problems.

A virus (considered here as the analog of a fault) penetrating our body will at first be latent (passive fault). Later, according to our physiological evolution, this virus can disturb the functioning of one organ (the fault is activated as an error in a module). Then, the infection may develop and propagate until it becomes an illness (propagation of errors and failure occurrence). The main detection mechanism is pain. Temperature is also used to detect an illness but it is fundamentally a protective mechanism. Here also, latency plays a major role by delaying the effects of the virus. The consequences of latency are positive. For instance, the illness is delayed for months or even years. However, it has also negative effects. For example, it may mask the presence of an infection that will develop inside an organ and will be very difficult to cure later.

4.4 EXERCISES

Exercise 4.1. Latency of an asynchronous counter

Let us consider a 16-bit asynchronous counter which counts pulses coming on the input *I* and gives the result on the output *O* with a natural 4-bit binary code (*a, b, c, d*). A stuck-at '0' of the variable *a* occurs while the counter is in the initial state (0, 0, 0, 0). Hence, the counter will now evolve

with constrained configurations (0, b , c , d).

Determine the average latency of this fault for a 2ms average-time between input pulses.

Exercise 4.2. Latency of a structured system

We want to analyze the latency of the 3-module structured product considered in Chapter 2 (represented again in *Figure 4.8*).

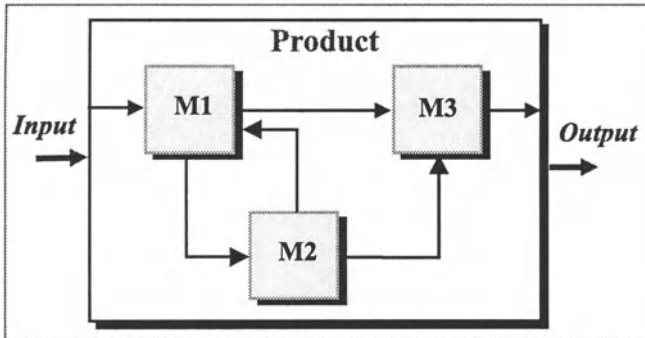


Figure 4.8. Structured system

Calculate the latency at the different levels of the structure by adding local latencies:

initial fault at time 0, latency L_1 at the level of module M_1 , propagation through M_2 , M_3 (latencies L_2 et L_3), and then failure at the output.

Numerical values: $L_1 = 10$ ms, $L_2 = 100$ ms, $L_3 = 30$ ms.

Exercise 4.3. Consequences of failures

A given product may be infected by one of the faults of a set of ten faults $\{F_1 .. F_{10}\}$ having the same probability of occurrence p . The maintenance contract which has been accepted for this application specifies that the repair time is equal to 4 hours.

An analysis of the application showed that the external consequences of the ten faults are as follows:

F_1, F_2 : benign consequence,

$F_3 - F_5$: significant with a 'fixed cost' of 5ku
(u being a given monetary unit),

$F_6 - F_7$: significant with a 'variable cost' of 6ku per hour,

$F_8 - F_{10}$: significant with a 'fixed cost' of 1ku plus a 'variable cost' of 3ku per hour of immobilization.

What is the average maintenance cost?

Exercise 4.4. Fault-Error-Failure in a program

The following procedure extract aims at counting the number of sheets in a book. At first, it computes the number of the last right page.

```
procedure Count_Number_of_Sheets (Sheets_Number:
                                out positive) is
  Last_Right_Page: positive;
begin
  Last_Right_Page := ...;
  Sheets_Number := (Last_Right_Page + 1) / 2;
end Count_Number_of_Sheets;
```

We assume that the expression which calculates the `Last_Right_Page` (noted ... in the program) contains a fault. We use this procedure in a book editing processor to analyze a book whose actual last right page number is 325.

1. Is there a failure if the faulty expression gives 326 as a result? Answer the same question with 327 and 328.
2. Is there an error in the three cases?
3. What do you conclude from this experiment?

SECOND PART

PROTECTIVE MECHANISMS

Confronted to the destructive mechanisms shown in the first part of this book, faults - errors - failures and their external consequences, designers, producers and users have developed numerous formal and empiric protection means. Their principles are introduced in this part. The associated practical methods and techniques will be studied in the third and fourth parts of this book.

We firstly introduce in Chapter 6 the three large groups of *dependability means* allowing faults and their internal and external effects to be mastered: *fault prevention*, *fault removal* and *fault tolerance*. The methods, which correspond to these approaches, are presented and organized according to several groups.

In Chapter 7 we consider the *dependability assessment* which allow us to measure the efficiency of the use of the previous methods. We study the *quantitative* approaches considering the principal criteria (*reliability*, *testability*, *maintainability*, *availability*, *safety* and *security*), as well as methods which permit us to evaluate them. We also discuss *qualitative* methods.

Redundancy is a fundamental notion which plays a major role in dependability techniques, principally for fault suppression and fault tolerance. In Chapter 8 we define this notion and its many diverse forms.

Chapter 5

Fault and Error Models

5.1 DEFINITIONS

5.1.1 Structural and Behavioral Properties

We have presented the mechanisms which transform faults into errors, failures and external consequences. We will now examine a fundamental question: how can faults and errors be expressed by means of mathematical modeling tools and what is the relevance of such models? The answer to this question is important because it affects the means that will be used to prevent, detect, or tolerate faults.

Faults were defined as adjudged causes of failures with regard to the system structure: a fault is a non-adequate alteration of the system structure. The identification of faults therefore depends initially on the modeling means used to express the studied system. Thus, for a given product, faults associated with a representation by state graphs are different from those associated with an electronic modeling based on MOS transistors. In the first case, a transition from a state to another may be a fault, whereas in the second case, it may be a short-circuit connecting two lines. In practice, numerous representations are used during the life cycle of a hardware/software product: from the behavioral model of its specification to the technological model of its implementation.

Moreover, the examination of each feature of a structure does not allow a precise fault prediction. We illustrated this fact by signaling that a connection linking two elements of a circuit can be considered as desirable (necessary to obtain a good functioning) or not (short-circuit). Similarly, the

simple reading of the program fragment 'if (A>B) ...' does not permit the conclusion that 'if (A>=B) ...' should have been written instead.

In fact, faults depend on the tool used to model the system, but also on the functionality that must be expressed by the system model. For instance, (A>=B) must be written instead of (A>B) because this last condition will not allow the expected behavior of the system to be obtained.

This discussion leads us to conclude that faults are as varied as the number of representation models and modeled systems! If an exhaustive list of all possible faults existed, it would certainly be huge. Consequently, what could be proposed to help design engineers in their fight against faults? To answer this question, faults must not be identified by their individual specificity but by common characteristics. Such common characteristics will allow fault classes to be defined, and thus techniques to be proposed to handle all faults of a given class. The issue is now the definition of such common characteristics.

As faults are defined as non-adequate modifications of a product's structure, we must express what is expected or required. This is generally difficult. Indeed, as the faults being considered in this book are unintentional, the knowledge of what is correct would prevent any fault creation during the development phases. Instead of precise expectations on the structure's characteristics, we will express intended *properties*. These properties are ordered into two main families:

- *physical properties*, also called *structural properties*, which deal with the static structure of the system or the product,
- *behavioral properties* which deal with the function performed by the system or the product.

5.1.2 Structural Properties

Properties of the first group, that is *physical/structural properties*, are expressed without any knowledge of the studied system's function. Where hardware technology is concerned, examples of such properties are: a wire connecting two components has been cut, components have not been mounted on a printed circuit board, or a board has not been properly plugged into a rack. Inspection tools can generally perform the detection of these faults. Concerning the domain of software technology, the properties deal with syntax faults such as a wrong keyword, a grammar fault, etc. A syntax fault is defined as the violation of properties on keyword and identifier sequences defined by the language grammar rules. Each property violation defines a class of potential faults and not one particular fault. For example, if an Ada compiler detects that a program calls a non-existent function, it

identifies a class containing numerous possible faults e.g. the called function is missing in the file, the name of the called function is wrong, or the called function specification has not been previously declared, whereas the body is written after the calling program.

A *fault model* defines a set of faults characterized by physical/structural properties, that is properties on the desired model structure.

The properties can be generic, that is to say not specific to any given system. *Generic properties* are related to the modeling tools, even if they are applied to each particular modeled system. For instance, the previous syntax fault class is characterized by the generic property: 'a called subprogram must be previously declared'. This requirement is a generic feature of the programming language. Each subprogram call can be at the origin of a specific fault belonging to this type.

5.1.3 Behavioral Properties

On the contrary, the second group of properties, that is the *behavioral properties*, characterizes faults by their effects as errors. For example, the occurrence of a '1' logical value at the output of an AND gate when a '0' value is applied at one input reveals an error. This error may be due to numerous faults: wrong operation assigned to the variables (e.g. a NAND instead of a AND), or wrong connection (e.g. the cut of an input). The property 'an assigned value must belong to the type range' defines a behavioral property for programs. It is a generic property as it is expected for all variables of any program. A particular error is raised when a variable assigned by a value does not belong to the range defined by its type. Thus, we put all the faults producing the same sort of errors together in the same class of an *error model*. These faults are characterized as violating the same behavioral property, that is, defined on the states intended to be reached when the system runs.

An *error model* defines a set of faults characterized as errors by a property on desired or intended states.

This distinction between fault model and error model is not always easy to establish. In particular, relationships exist between the two notions. For instance, in the case of the preceding AND gate error model, a more precise investigation would discover a physical fault such as a short-circuit between the output line and the ground line, that is, a fault model. In the literature, most authors speak of fault model as a generic term. The term *error typology* is also appropriate.

The variety of the means used to model the systems in addition to the variety of expressed properties do not allow an exhaustive list of *fault/error models* to be provided. Our presentation focuses on examining faults and errors relevant to the last phases of the development process, which are associated with the implementation technologies. We examine the families of fault/error models relevant to *hardware* and *software technologies*.

In section 5.2, we present examples of significant fault/error models of electronic circuits and software programs. We then discuss their relevance in section 5.3. We analyze, in section 5.4, some faults altering two simple examples: functional and hardware faults affecting a hardware addition circuit (section 5.4.1), and faults of a small program and of its runtime resources which support its execution (section 5.4.2). These pieces of information will be useful to understand the protection mechanisms explained in the following parts of the book.

5.2 SIGNIFICANT FAULT AND ERROR MODELS

5.2.1 Faults and Errors at Different Representation Levels

Table 5.1 presents some classes of faults and errors for modeling means used at different steps of the design of an electronic product.

Level	System examples	Examples of types of faults or errors
Abstract	Finite State Machines Petri nets	Erroneous states / arcs (fault)
Program	Programming languages	Bad using of statements, Erroneous constants (faults)
Functional	Register Transfer Languages	<i>Functional faults/errors</i> affecting the modules (flip-flops, registers, etc) <i>Control faults/errors</i> affecting transfers between modules
Logic	Gate, Flip-Flop	Stuck-at 0/1 of gate inputs/outputs (errors)
Electronic	Transistor MOS	Stuck-at 0/1 of wires Short-circuit, open lines (faults) Coupling (errors)
Technology	Layout Physical structure	Default on the layout Erroneous dimensions of technological elements (faults)

Table 5.1. Faults and errors at various representation levels

These 6 levels illustrate the significant steps of the design process: abstract, program, functional HDL (Hardware Description Level), logic, electronic and technological. Then, in *Figure 5.1* we provide some simple examples of faults and errors for these six levels.

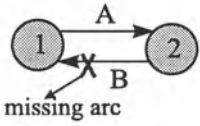
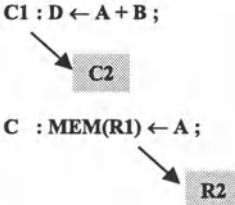
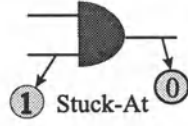
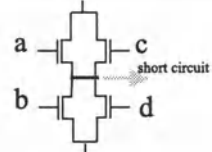
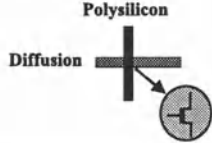
<p>Abstract: design fault</p> <p>Arc 2 → 1 was forgotten If input <i>B</i> is applied from state 2, the system stays in this state</p>	
<p>Program: design functional fault</p> <p>Line 1 becomes: if $A < B$ then The result <i>S</i> is not incremented when $A = B$</p>	<pre>if A <= B then S := S + 1; end;</pre>
<p>HDL: functional and control faults/errors</p> <ol style="list-style-type: none"> When condition <i>C1</i> arises, the result of $A + B$ is stored in register <i>D</i>. If <i>C2</i> is written instead of <i>C1</i>, an error in the <i>synchronization</i> of this treatment occurs When condition <i>C</i> arises, the value of register <i>A</i> is stored in the memory at the address provided by <i>R1</i>. If <i>R1</i> is replaced by <i>R2</i>, then an addressing error exists 	<pre>C1 : D ← A + B ; C2 : MEM(R1) ← A ;</pre> 
<p>Logic: hardware faults</p> <p>Stuck-at '0' or at '1' of the input/output of the gates. For instance, stuck-at '0' of an output or stuck-at '1' of an input</p>	
<p>Electronic MOS: hardware faults</p> <p>Short circuit between two wires: the logic function of the MOS network is modified</p>	
<p>Hardware technology: design fault</p> <p>Creation of a parasitic MOS transistor due to an involuntary crossing of two connections (polysilicon and diffusion)</p>	

Figure 5.1. Fault examples at several abstraction levels

We need means to identify the numerous faults which can affect the structured models expressed during development phases. As previously mentioned, due to the large range of possible faults and due to the difficulty involved in considering a fault as the presence or the absence of structural elements, the faults could be classified according to the errors they generate.

Unfortunately, we are again in a similar situation: the possible errors are innumerable and they strongly depend on studied products. For instance, when software applications are concerned, properties specific to each application can be expressed correlating the internal state values:

- relationships between the values of the variables of the program, for example the value of X must be located between those of Y and Z ,
- or relationships on the sequencing, for example a subprogram $P1$ must be called before $P2$.

The proposal of an ‘universal’ classification of the faults therefore requires *expected properties which depend only on the system’s modeling means*, that is, independent of each specific modeled system. For instance, if the programming level is considered for a software system, the following property is generic:

the values assigned to a variable must belong to the set defined by the type associated with the variable.

All the program faults leading to this *generic error* belong to one class of the error model. We insist again on the fact that we defined an error model and not a specific error: each occurrence of an ‘out-of-range’ assignment of a given variable is one error; hence, being valid for any variables, the property defines an error model.

The following two sections are dedicated to the study of some significant fault and error models concerning the hardware technology (section 5.2.2) and the software technology (section 5.2.3).

5.2.2 Hardware Fault/Error Models

5.2.2.1 General Error Models

Hardware fault effects can be characterized independently from any precise system modeling tool. For this reason, this error modeling approach is qualified as ‘general’. Five independent properties that take two opposite states are generally used to classify errors at this level:

- logical or non-logical,
- static or dynamic,
- permanent or temporary,
- single or multiple,
- symmetric or asymmetric.

Logical errors are characterized by transformations of logical values: '0' becomes '1' and vice versa. On the contrary, **non-logical errors** provoke alterations of the logical levels outside the specification domains. The altered signals take a value between '0' and '1'.

Static errors correspond to stable undesirable state, for example a gate output '1' instead of '0'. **Dynamic errors** represent faults which provoke transient undesirable states, for example an oscillation of a signal before reaching a correct stable value.

Permanent errors affect the functioning for a long time or in a definitive way, whereas **temporary errors** have limited operation duration (sometimes very short). Concerning this notion, another set of definitions is found: **hard errors** corresponding to permanent errors and **soft errors** corresponding to temporary errors resulting from external causes (*transient faults*). These terms are used to characterize some errors altering RAM (mainly Dynamic RAM) for memory testing. Soft errors can be induced by cosmic rays or alpha particles.

Single errors disturb only one element (for instance a transistor), when **multiple errors** disturb the functioning of several elements (for instance a problem in the electrical supply circuit affects all the components). The **order** of a multiple error is the number of elements which are altered.

Finally, **symmetric errors** provoke state modifications with the same probability (for instance, '0' to '1' and conversely), whereas **asymmetric errors** have not the same probability to switch to '0' or to '1': for example, several lines are switched to value '1' due to the energy provided by an external particle disturbing a component.

These five parameters taking two values, we define ten error classes. Thus, logical, static, persistent, single and symmetric errors identify a class of faults leading to these kinds of errors. Note again that these properties are actually generic as they are independent of the modeling tool and thus of the analysis level considered and also of the modeled system. From this first characterization of the properties, more precise classes of faults will be defined taking the system modeling tool into account.

5.2.2.2 Error Models at Logical Level

The expression of a system using a model at logical level is based on interconnected gates. We assume that these gates implement the following elementary logical functions:

AND, OR, NOT, NAND, NOR, EXCLUSIVE OR (noted **XOR**).

This level is then extended, handling more complex objects such as multiplexers, decoders, latches, registers, counters and other simple functions to store or process data. Thus, the HDL level is reached. At this

level, the fault effects are characterized by alterations of internal states of components or alteration of signals on wires linking logical components.

All technological faults affecting gates are observed at their input and output level. A very old but still popular error model is the *single stuck-at 0/1* model. This is defined as an input / output wire (or *node*) having a persistent '0' or '1' assignment.

Figure 5.2-a) illustrates this model for a two-input AND gate. Each one of the 3 input and output ports can be stuck at '0' or '1'; hence there are 6 persistent single logical errors. Two of them are represented in the figure (stuck-at '0' of *b* and stuck-at '1' of *c*). By generalization, we define a *multiple stuck-at fault* as any set of single stuck-at faults, for example the stuck-at '0' of line *b* and the stuck-at '1' of line *c* in Figure 5.2-a).

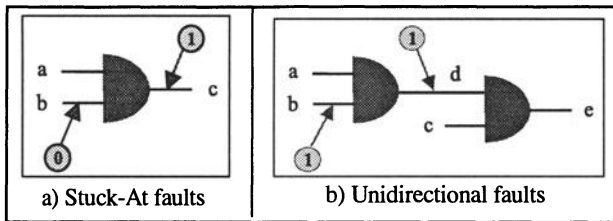


Figure 5.2. logical faults/errors

What we identify as 'stuck-at fault' represents in reality a set of unknown technological faults which can alter the electronic components inside the gate. This set of faults is strongly dependent on the technology used to realize the gate. We could also say that the 'stuck-at' model is an *error model* corresponding to the activation at the input/output level of internal 'real' faults. This apparent divergence of interpretation is a matter of observation about the gate module. It has no influence on any propagation analysis outside the module.

A *unidirectional error* is a multiple asymmetric error such that all altered lines are stuck at the same value. An example is given in Figure 5.2-b): two lines, *b* and *d*, are stuck at the same value '1'. This model is realistic when expressing situations like a power failure of a MOS circuit or a line cut in a Bus connecting several units or else a parasitic phenomenon altering a transmission line.

5.2.2.3 Fault Models at MOS Switch Level

At the MOS switch level representation, a circuit is expressed as a network of interconnected MOS, each one being considered as a simple

switch whose state is ON (conducting) or OFF (blocked) according to the signal applied to its gate input.

This model is naturally closer to the physical reality than the logical or the HDL levels. Hence, the related fault models are more realistic, that is to say closer to the likelihood of electronic faults which disturb the components. Unfortunately, these fault models are also much more complex in terms of number of faults.

In the two following sub-sections, we will examine some examples of fundamental fault models, first at MOS network level, then at MOS gate level. Two kinds of faults are specific to this level of representation:

- *shorts* between lines,
- *opens* paths on conducting lines.

A particular case of a short fault, called a *bridging fault*, implies a ‘wired logic’ functioning. This kind of faults can occur in certain gate structures when the output lines of two gates are accidentally connected together. Then, the resulting signal on this line is the logical OR (or the logical AND according to the technology) of the fault-free output lines.

Elementary fault models at switch level

A very simple and basic MOS fault model supposes that each line can be stuck-at ‘0’ or ‘1’, and each transistor can be:

- *stuck ON* when the transistor is always conducting,
- *stuck OFF* if the transistor is always open (no conduction).

The fault is ‘single’ if only one element is altered. On the contrary, it is ‘multiple’ if several elements are faulty. In *Figure 5.3*, fault *F1* (transistor *T1* is OFF, hence its equivalent switch is open) is an example of such a single fault.

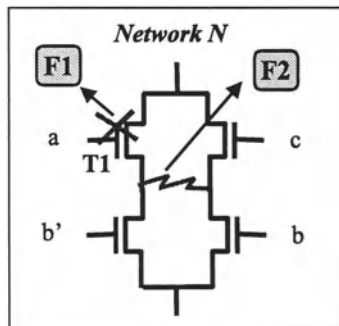


Figure 5.3. Faults of a MOS network

Without fault, this network implements the logical function $R = a.b' + b.c$, where the symbols '+', '.', and '' represent the logical functions OR, AND, and NOT. When fault $F1$ occurs, the function becomes $R1 = b.c$ which produces a wrong output for two input vectors (a, b, c) : 100 and 101.

This basic model can be improved by adding *shorts* between lines. For example, fault $F2$ of the network of *Figure 5.3* is such a short. The resulting function of the network is: $R2 = (a + c).(b' + b) = a + c$.

One can easily show that the modified function $R2$ is different from the good one R . Exercise 5.1 analyzes in greater detail the influence of switch level faults on the behavior of this network.

Fault models at CMOS gate level

CMOS electronic circuits are generally organized as structures comprised of two wired networks: a Pull-Up network connected to the '+' power line and the Pull-Down network connected to the '-' power line (*Figure 5.4*). According to the input values, either the Pull-Up network is conducting and the Pull-Down network is blocked, forcing the output to a '1' value, or the Pull-Down network is conducting and the Pull Up network is blocked, forcing the output to a '0' value.

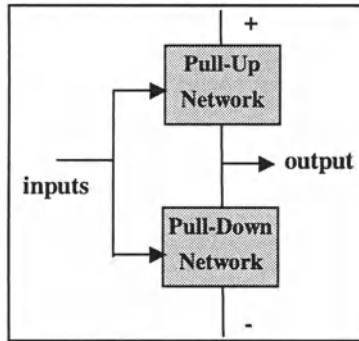


Figure 5.4. CMOS gate structure

Figure 5.5 shows two simple fault examples affecting a NAND gate:

- $S1$, short-circuit in a transistor, is equivalent to a stuck-at '1' of input a ,
- $S2$, short-circuit in a transistor, is equivalent to a stuck-at '0' of input b .

The resulting two faulty functions, f_{S1} and f_{S2} , are compared to the normal one f in the truth table of *Figure 5.5*:

$S1 \rightarrow f_{S1}$: this fault can be detected by the input vector $(a b) = (0 1)$,

$S2 \rightarrow f_{S2}$: this fault can be detected by the input vector $(a b) = (1 1)$.

Unfortunately, many physical faults cannot be reduced to such simple logical transformations. In some cases, the altered circuit can even be transformed into a sequential circuit, as presented now.

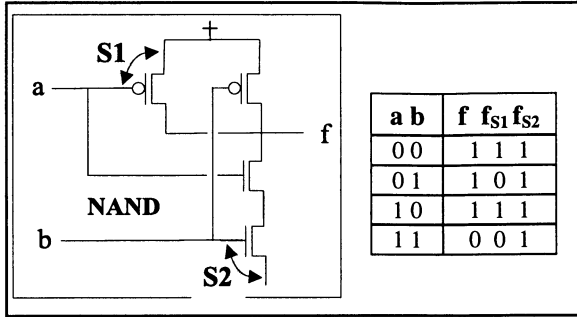


Figure 5.5. Faults of a NAND MOS circuit

Stuck-Open Model. The *stuck-off* or *stuck-open* model considers that a transistor is blocked in the OFF state. An example is given by fault *O* of Figure 5.6 a). In this case, if we apply the input vector (01), the two Pull-Up and Pull-Down networks are blocked (not conducting) and the output value depends on the preceding state of the circuit because of its output equivalent capacitance. Hence, this circuit becomes sequential.

The exhaustive input sequence <00, 01, 10, 11> does not detect this fault. To detect the fault, it is necessary to apply the two ordered vectors <11, 10>: then, *f* remains at value '0' instead of going to '1'.

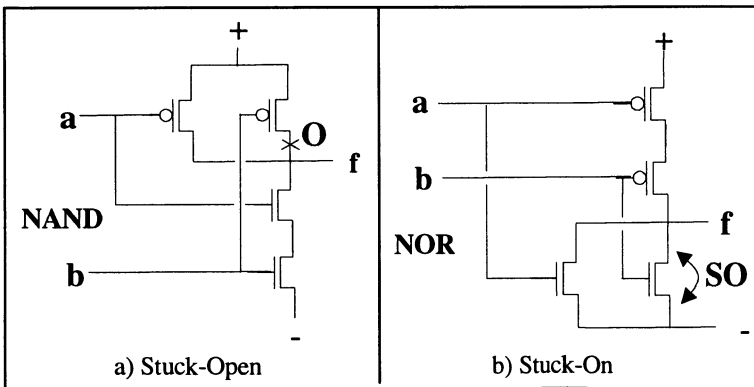


Figure 5.6. Stuck-Open and stuck-On faults in a NAND circuit

Stuck-On Model. The *stuck-ON* model considers that a transistor is blocked in the ON state. An example is given by fault *SO* of Figure 5.6 b).

In this case, if we apply the input vector (00), the value of output f is not defined because both networks are conducting, producing an electrical conflict between the power supply and ground lines. To reveal this fault, specialists traditionally use the *IDDQ testing* method which consists in measuring the power supply current passing through the circuit during quiescent states.

Short-Circuit Model. The *short-circuit* model considers that two outputs of gates are wired together. This fault category also creates situations where the outputs cannot be easily specified to '0' nor '1'.

Other fault models. The simple fault models previously introduced can be completed by taking into account other considerations such as the notion of 'electrical force' or 'response time' of the components. The *temporal (or timing) fault/error models* deal with incorrect response time of components: they are more realistic but much more complex to implement, in particular for test applications. However, some timing fault can be actually induced by crosstalks between wires. A *delay fault* occurs when a signal propagated through a circuit is slower than it should be. This term defines a specific *temporal fault*. In many cases, delay faults do not affect the functioning of a device, but merely skew the results in time. This type of performance change can, however, be totally unacceptable if it causes the violation of timing specifications.

5.2.2.4 Error and Fault Models at Technological Level

At the technological level, faults can be analyzed with more accuracy. We mention hereafter some significant classes of faults or errors.

- Defects of the crystalline structure of the Silicon; the quality of the wafers which are at the origin of all integrated circuits can be altered by a certain number of crystalline structure defects: area defects, line defects, or spot defects (fault model). These defects may induce cuts, shorts, etc.
- Punctual or global fabrication defects due to dust, optical or chemical fabrication problems, etc. (fault model).
- Defects occurring during the use, e.g. aluminum electro-migrations, line cuttings, layer-to-layer shorts, thin oxide shorts (MOS gates), floating nodes, soldering defects (fault model).
- Aggressions during the use: parasitic signals due to crosstalk between wires or external electro-magnetic induction, alpha-particle contamination, X-ray action, etc. (error model).

5.2.2.5 Other Integrated Circuit Faults

The preceding fault/error models cannot directly express a great number of faults that alter electronic components. However, these actual faults have to be considered during the component's production stage. We provide some examples of such faults that have mechanical, chemical or other nature.

- Encapsulation: Integrity, waterproofness, traces of welding.
- Package: integrity, support quality, dimension, leaks, thermal and mechanical resistance.
- Internal cavity: humidity, free particles, and quality of the support.
- Contact points: spacing, short-circuits, passivation remains, scratches.
- Die: corrosion, width, short-circuits, scratches and holes, alignment, cover, chemical defaults, passivation.
- Electric quality: continuity and short-circuits, electric stability and characteristics, temperature performances and stability, sensitivity to electrostatic discharges, sensitivity to radiation, design quality.
- Pins: strength, aptitude to welding, airtightness, materials and finishing, resistance to heat, resistance to humidity, spacing and length, damages.
- Support of the die's fixing: strength (adhesion), consistency - uniformity, cover, humidity, die orientation, excessive accumulation of materials, free particles, dissipation, thermal and mechanical constraints, support type, re-processing, general manufacturing quality.
- Connection wires: strength, position, height and curve, spacing, adhesion, uniformity, size, current density, re-processing, metallic contamination, thermal constraints, cuts.

5.2.3 Software Fault and Error Models

In this sub-section, we consider fault and error models relevant to software technology. We firstly present a general error model. Then we discuss fault and error models at *source code* level. Finally, we introduce error models at *executable code* level.

5.2.3.1 General Error Models

The first general error models introduced in section 5.2.2.1 for hardware systems are also applicable to software technology. In practice, three of the five parameters presented are relevant for this technology: static or dynamic errors, persistent or temporary errors, and single or multiple errors.

Static or dynamic errors. Let us consider a system which handles sampled data coming from a sensor. The new input value is stored in a variable x every 10ms. If the program that implements this sampling function puts a *null* value in x at the end of the using of the last sampled value (e.g. for process control purpose), the variable x contains a wrong value *null* till a new value is sampled and stored. This is a dynamic error as the final value of x is correct. An example of static error would be an erroneous Analog-Digital conversion that would store a wrong static value.

Permanent or temporary errors. To illustrate this property, we now consider a multi-task program. A consumer task $T1$ treats a shared variable x assigned by a producer task $T2$. When the program execution starts, x is not assigned to a pertinent value. If $T2$ assigns a correct value to x before $T1$ reads it, no problems occur. On the contrary, if $T1$ reads the value before it has been assigned by $T2$, an error occurs. If the tasks $T1$ and $T2$ are cyclically executed, this error disappears at the next cycle (after x has been assigned by $T2$). On the contrary, if x is read by $T1$ only once, the error is permanent.

Single or multiple errors. Single errors are errors that affect only one element of the software product. The term 'element' depends on the model used. At programming level, it may be a variable, a subprogram, a package, etc. At design level, it may be an object, a resource, etc. On the contrary, multiple errors occur when several elements are affected.

5.2.3.2 Fault Models at Source Code Level

The syntax of a programming language is defined by rules of a grammar. A grammar does not specify a unique sequence of keywords or identifiers. Otherwise, only one program structure could be written. The rules define constraints, that is properties, on the acceptable sequences. For instance:

```
<statement> ::= <if-statement> | <loop-statement> | ...
```

enumerates the various possible statements, by using the standard BNF (Backus-Naur Form) representation.

Thus, the first role of a compiler is to check that the compiled program is in accordance with the properties required by the programming language grammar. Any violation of such a property highlights a fault.

Consequently, each programming language defines a fault model. Therefore, the choice of a programming language must take into account the features it offers, but also the fault model it provides, that is the verification actions processed by its compilers.

Let us consider the following C language extract:

```
if (A>B) A**;
```

```

if (A>C) C++;
else B++;

```

The syntax of the program does not impose syntactic constraints on the definition of the beginning and the end of programming blocks. Hence, the previous program may be erroneous, the expected correct program being:

```

if (A>B){
  A++;
  if (A>C){
    C++;
  }
else {
  B++;
}
}

```

On the contrary, the use of the Ada language makes necessary the expression of the end part of the statements (end if). For instance, the preceding algorithm becomes:

```

if (A>B) then
  A:=A+1;
  if (A>C) then
    C:=C+1;
  end if;
  else
    B:=B+1;
end if;

```

Note. The else part is not required by the syntax but it is strongly recommended by *Quality Guidelines*. When nothing is to be done, a null block must be used:

```

... ..
else
  null;
end if;

```

5.2.3.3 Error Models at Source Code Level

As previously mentioned, a program is a structure made up of an assembly of features provided by a language. These features are defined by: their *syntax*, allowing fault models to be expressed, and their *semantics* specifying their behavior.

Such as in electronic technology, we are searching for expected properties which must be generic, that is, applicable to any program

whatever its functionality may be. The negation of properties associated with the semantics of the programming language features therefore defines such an error model. We present five examples.

1. *A function does not return a value.* This is an actual error of the program execution. Indeed, unlike the other subprograms called *procedures*, each *function* must return a value to conclude its execution. Such a property is defined by the semantics of the used programming language. As previously, we do not want to express the fault which is at the origin of this error. It is maybe a simple case of negligence: the author forgot to write the statement `return X;`. This statement may exist, but another fault in the previous statements leads to a control flow path which is not concluded by the execution of this return statement.

2. *An input parameter of a subprogram is not assigned by an actual value at subprogram call.* This error occurs if `Push(X);` is called with a non-initialized value of `X`.

3. *An output parameter of a subprogram is not assigned at the subprogram body execution completion.* For instance, no value is returned in `Y` after the execution of `Pop(Y)`.

4. *A variable whose type is constrained is assigned by a value not belonging to the range specified by this type.* Example:

```
subtype Ice_Temperature is integer range -70 .. 0;
Freezer_Temperature: Ice_Temperature;
. . .
Freezer_Temperature := . . . expression . . . ;
```

where the expression evaluation returns +270 at runtime.

5. *A first task calls the service of a second task which does not exist.* This occurs when the second task was not previously created or if, when being created, it was then achieved. The potential faults which are at the origin of this last error are numerous:

- the source program design explicitly expresses that the second task must be completed before the call,
- the second task was achieved due to an error raised during its execution,
- the second task was unintentionally killed by another task.

5.2.3.4 Error Models at Executable Code Level

As usual, the error models highlight the violations of expected properties. However, the properties now concern attributes of the executable code.

The fact that *the execution of a called subprogram is not terminated by an instruction 'return'* is such an example. This instruction is absolutely necessary to pop the call context in order to restore the caller context. Numerous causes can be at the origin of this error. For instance, it may be due to the execution of a jump instruction of the subprogram body whose associated address was corrupted. The fault which has provoked such a situation may be:

- a bad expression used to calculate the branching address due to a compiler failure,
- a bad constant address coming from an electromagnetic disturbance of a bit of the memory word where this data is stored, etc.

The execution stack overflows is a second example of error model at executable code level. A stack is used at runtime to manage subprogram calls (for instance, local variables and return addresses are temporally stored), to handle interruptions, etc. Here again, various faults can be at the origin of this class of errors such as:

- infinite recursivity of a subprogram due to bad design or programming,
- bad assessment of the stack memory size requirements due to:
 - the compiler whose generated code does not optimize the stack use,
 - the runtime executive (operating system) which does not master correctly the dynamic memory allocation.

5.3 FAULT AND ERROR MODEL ASSESSMENT

We introduced the most significant fault and error models used for hardware and software technologies. Of course, these models are very different from one another as they reveal different realities (hence, different system models). Each one makes assumptions about the possible faults errors that may affect the system's representation or its operation. Moreover, they often present a probabilistic character coming from statistical data or measurements from samples. The fault occurrence probability is an important piece of information accompanying the qualitative and technical characteristics of this fault.

5.3.1 Assessment Criteria

The engineers using these models for dependability purposes have to answer the question: "How can I choose the best fault/error model and how

do I assess its relevance and efficiency?”. Even if we cannot provide precise grades to each model, several criteria to *assess* them can be established. We have identified 6 main criteria which allow the evaluation of the quality of fault and error models: relevance, fault expression capability, fault partitioning, distribution equilibrium, genericity / specificity and tractability.

1. Relevance

At first, there is not one good model, as each one is associated with a development phase and therefore is appropriate in order to characterize the faults and errors occurring or acting at this phase. For example, a physical defect of a MOS transistor will certainly provoke a dysfunction of the microprocessor which contains this transistor and finally lead to an error at the application software level. However, an error model at software level may be able to detect the resulting error, but not to diagnose the faulty MOS. For this diagnosis purpose, a switch level fault model is more suitable! The ‘realistic’ notion of a fault model reveals its accuracy to identify faults. The more precise the model is, the more pertinent and efficient the fault analysis is. A fault not expressed by the model cannot be interpreted, and the protective means used to detect and handle it can be inefficient.

2. Fault expression capability

A fault/error model defines classes of faults. Such a model has a high fault expression capability if it allows a great number of faults to be characterized. Each fault belonging to this set of faults is identified in at least one *fault/error class*.

3. Fault partitioning

A third characteristic is the fact that the model makes a mathematical *partition* of the considered faults into several classes. This means that each fault is identified in exactly one class. Otherwise, faults are difficult to identify precisely.

4. Distribution equilibrium

A good model must induce an equilibrated distribution of the considered faults into its classes. This property makes the fault diagnosis easier.

5. Genericity or specificity

In order to be used in many different contexts, a fault/error model must be generic. Even if they are applied to specific systems, fault/error models depend on the modeling tool used to express the system (gates, programming languages, Petri nets, etc.), but they do not depend on the modeled system.

For example, the ‘stuck-at 0/1 fault model’ is based on a gate level representation and may be applied to any logical structure, independently of any final technological implementation. However, even if these fault models are useful, they do not allow the handling of numerous errors specific to the particular functionality of each system. To illustrate this point of view, we consider the following subprogram extract:

```

procedure Min_Max(L: in List; Minimum, Maximum: out
                    Element) is
. . .
begin
. . .

```

We assume that this subprogram returns the values Minimum and Maximum of the elements of a given list (L). Certain presented fault models assume that a list can be unassigned at call-time or that the subprogram body execution can omit to return values in parameters Minimum and Maximum. However, these fault models are not specific to this subprogram. They only depend on the semantics of the language features ‘in’ and ‘out’.

On the contrary, ‘the Minimum value is lower or equal to the Maximum value’ is a specific property: it depends on the functionality of the considered subprogram.

6. Tractability

Finally, the fault/error models must be tractable, as automatic or semi-automatic tools generally use them. Unfortunately, certain models that are technologically pertinent are often unrealistic for practical reasons. For instance, the CPU time consumed by a computer or the memory size that is required by the tools to analyze or to simulate these fault models or to generate test sequences is prohibitive. So, in order to use tools, we often make many restrictive assumptions such as ‘the logical faults are permanent and single’, whereas the reality shows that most of the faults are temporary. A rate of 80% of temporary faults in electronic systems is often given.

5.3.2 Relations Between Fault/Error Models and Failures

As we noticed, different fault/error models can be associated with a modeling tool of a system, and it is very important to measure their relative significance according to their use (such as testing). *Figure 5.7* illustrates this notion. For a given modeling tool, the normal functioning is represented by a sub-set F of all theoretically possible behaviors (universe U). In the general case, two different fault/error models reveal two sub-sets (noted $EM1$ and $EM2$) of this universe. Some failures belonging to this universe

(and caused by unknown faults) are represented by both fault/error models, some failures are represented by one model only, and some other failures are not represented.

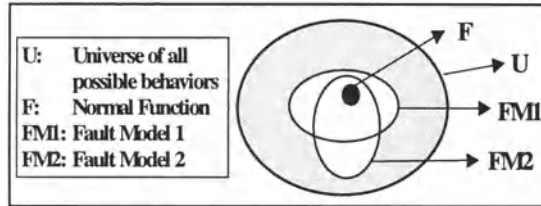


Figure 5.7. Fault models and failures

Different and various modeling tools are used along a project development, involving various fault/error models. The pertinence and the consistency of these models is a real issue.

To highlight these notions, we will consider a very simple logical gate circuit and show some relations between the fault/error model and the resulting failures.

Example 5.1. Fault/error models and failures

Let us consider the system of *Figure 5.8* represented at gate level. This correct system expresses the logical specification $f = a.b + c$. If we assume that the theoretical behavior is combinational (restrictive hypothesis), the set of possible correct and incorrect logical functions is constituted of 1 correct function and $2^8 - 1 = 255$ incorrect functions. So, this circuit has 255 different failures in the combinational universe.

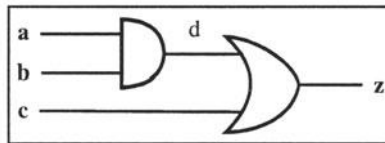


Figure 5.8. Redundant wire

As a first fault model, let us consider the ‘single stuck-at 0/1’ fault/error model introduced in 5.2.2.2. As there are 5 lines (gate inputs/outputs), this model represents 10 faults. We want to deduce the failures associated with each fault. Analytic and structural methods allowing to do this will be explained in other chapters of this book. Here, for this very small circuit, a simple method (formal analysis) consists in expressing the logical function corresponding to each fault and draw the truth tables. For example, if input b

is stuck at 1, the function becomes: $z = a.1 + c = a + c$. If we apply this method to all faults, we observe that several faults produce exactly the same failures: the stuck at 0 of lines a, b, d , and the stuck at 1 of lines c, d, z . Thus, we find 6 different erroneous functions (classes of failures). The important consequence of this analysis is to show that 249 theoretically incorrect functions, hence failure configurations, are not covered by this fault model!

Now, suppose that a functional fault has transformed the AND gate into a NAND gate. This new fault does not belong to the previous stuck-at fault model. This fault provokes a new incorrect function, so a new failure class. Many failures cannot be provoked by any of these two fault models. So, the question is: *can all theoretically possible failures occur?*

The answer to this question depends on the reality of the faults that might occur during the stage considered. The good news is that, according to actual used technologies, most failures have a very small occurrence probability. Exercise 5.3 proposes a deeper analysis of this example.

5.4 ANALYSIS OF TWO SIMPLE EXAMPLES

To conclude this chapter, we consider some functional and technological faults altering two simple examples and analyze the failures they imply. The first example is a full adder described at gate level, and the second example is a small program which computes the average value of a set of numbers.

5.4.1 First example: Hardware Full Adder

5.4.1.1 Specification and Design of the Circuit

The examined circuit is an academic three input ‘full adder’ whose specifications are given by *Figure 5.9*:

- the output S is the ‘modulo 2’ sum of the three input bits (‘exclusive or’ of these bits, noted \oplus : $S = a \oplus b \oplus c$),
- the output C is the carry of the input bits, i.e. the Majority function ($C = a.b + a.c + b.c$).

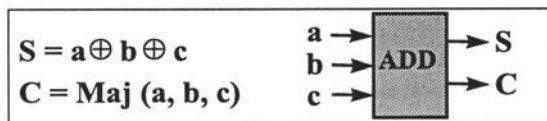


Figure 5.9. Specifications of the Full Adder

A modular design of this circuit at functional level uses two modules 'half-adders' (noted 1/2A) connected as shown in *Figure 5.10* where the apostrophe ' represents the logical complement. Each half-adder is made of an XOR gate ($T = a \oplus b$) and a NAND gate ($U = (a.b)'$). The resulting logical circuit has two XOR gates and three NAND gates (*Figure 5.11*).

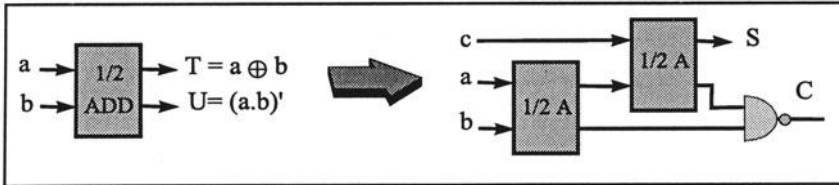


Figure 5.10. Design of the adder at functional level

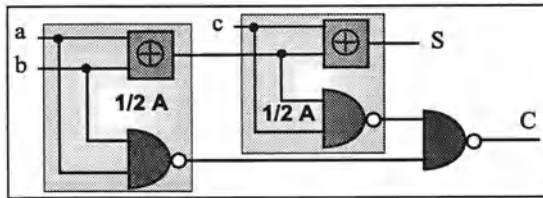


Figure 5.11. Resulting gate circuit

5.4.1.2 Fault Examples

We will examine the two examples of faults of *Figure 5.12*: a functional fault (fault 1) and a technological/hardware fault (fault 2).

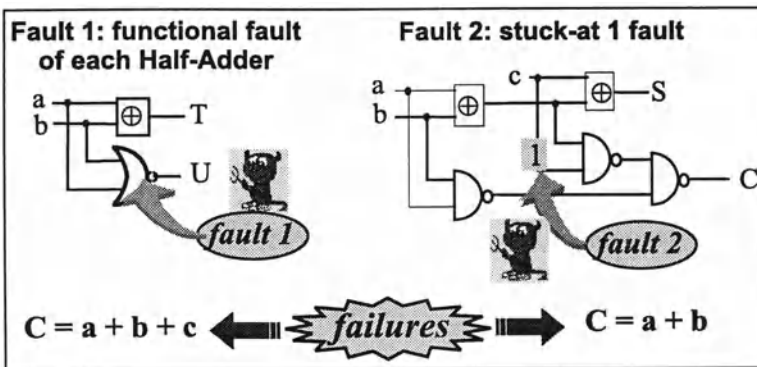


Figure 5.12. Fault examples of the adder

Functional fault (fault 1)

During the design phase of the half-adder, the correct NAND function ($U = (a \cdot b)$) has been erroneously transformed into a NOR function ($U = (a + b)$). As a consequence, this fault affects both half-adders and can produce an error at the U output of one module or both, depending on the input values (a, b, c). These errors cannot be propagated towards S . On the contrary, they disturb the C output variable which becomes a OR function ($R = a + b + c$) instead of the desired Majority function. Hence, a failure occurs for three input vectors: (0 0 1), (0 1 0) and (1 0 0). For each case, C provides the value '1' instead of '0'.

It should be noted that this fault belongs to a class of functional faults that alter the gates of the half-adder module. In general, it may be very difficult to identify all the faults belonging to this class.

Hardware fault (fault 2)

The circuit carried out is now supposed to be correctly designed. Let us imagine that during the operation of this circuit, one NAND input (*Figure 5.12*) is stuck at '1'. This fault will be activated as an error each time a '0' value is expected to arrive at this point. This error has no effect on output S . On the contrary, it can be propagated towards output C and it provokes a failure for two input vectors, (0, 1, 0) and (1, 0, 0), giving a wrong '1' value instead of '0'.

Let us note that this fault belongs to the stuck-at error model of the global gate structure of the adder. This model identifies 30 classes of faults as the resulting gate circuit given in *Figure 2.10* has five 2-input gates (hence, the number of errors is $5 \times 3 \times 2 = 30$).

5.4.2 Second Example: Software Average Function

Functional faults of software are introduced during the creation phases: specification, design and production. They are numerous and difficult to model. Moreover, during the operation phase they are mixed with technological faults coming from the executive context of the programs: software environment (runtime executive, etc.) and electronic components (micro-controller, I/O interface, etc.). As an example, let us consider an average function implemented in a control system.

5.4.2.1 Faultless Program

The faultless program computes the average value of a set of N elements previously stored in an array A . If A contains the float numbers (-12.0, 3.0, 26.0, -4.0), the program gives the correct result 3.25.

```

function Average(A: in Set) return Element is
Sum: Element :=0.0;
begin
  for I in A'range loop
    Sum := A(I) + Sum;
  end loop;
  return Sum / (A'last - A'first + 1);
end Average;

```

5.4.2.2 Fault Examples

We will firstly consider three functional faults; then, we will examine some technological faults induced by the hardware and/or software environment of this program.

1. Functional faults

Fault 1

A fault introduced during the programming affects line 5 which becomes

'Sum := A(I) - Sum;' (see *Figure 5.13*).

The programmer keypressed a '-' character instead of a '+' character. This very simple mistake of one character totally modifies the result. With the previous values we obtain $\text{Average} = -3.75$, instead of $+3.25$

Except for very special configurations of the numbers stored in *A* (e. g. if they are all null), this fault is activated and the function fails. If, for example, the function is used as a filter in a flight control system of an aircraft, the consequences of this fault can be catastrophic for the mission.

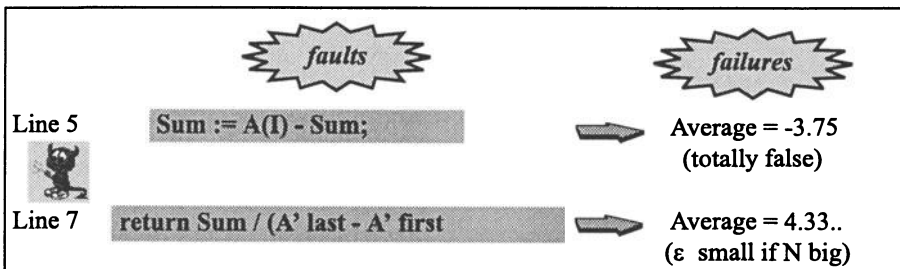


Figure 5.13. Specification/design faults

Fault 2

Let us now consider a second fault that alters line 7 as follows (*Figure 5.13*): `return Sum / (A'last - A'first);`

This fault is not activated into an error only if the sum of all elements is equal to zero. In fact, the average is computed by dividing the sum of all elements by their number minus 1. From the initial numerical values, the result is $Average = 4.33\dots$. The error seriousness depends on the number of elements: it will be small if this number is high. Hence, the consequences of the fault on the mission can be small or important according to the specific case.

Fault 3

We assume that the declaration of variable `Sum` does not include an initialization to 0.0. Therefore, the returned value is:

$$(Init + A(1) + \dots + A(N)) / N,$$

where *Init* is the value located in the memory when `Sum` is allocated.

Due to this fault, the returned result is hazardous, that is to say, it depends on the actual value of *Init* at each function call.

Error models

After the presentation of these three fault examples, we now study the error models that allow the faults to be characterized.

Concerning fault 1, let us assume that the following constraint exists on the type of 'Element':

```
subtype Element is float range 0.0 .. 10.0;
```

This constraint leads to a first error model. The first fault assigns to `Sum` a negative value if the first set is $A = (4.0, 3.0, 2.0, 1.0)$. The returned value is $-2.0 / 4.0 = -0.5$ which does not belong to the range 0.0 .. 10.0. An error will be raised when the value will be returned.

Therefore, the definition of a constrained type seems to be an efficient error model. However, this does not mean that it will always highlight the presence of faults! As an example, let us consider the second set of values (1.0, 2.0, 3.0, 4.0). The returned value is $2.0 / 4 = 0.5$, which is included in the defined range 0.0 .. 10.0.

Fault 2 leads to an error if the following equation is true: (sum of the N elements) / ($N - 1$) ≥ 10.0 , that is, (sum of the N elements) $\geq 10.0 \times (N - 1)$.

For the two previous sets, the sum of the $N = 4$ elements is 10.0, which is less than 30.0, so no errors will be detected, even if a fault exist.

To characterize fault 3, an error model is defined by the following property: "A variable used to evaluate an expression must possess a previously assigned value".

Thus, the first loop execution will process the statement '`Sum := A(1) + Sum`', and will raise an error, as the right side term `Sum` is undefined.

2. Technological Faults

Let us now suppose that no specification and design faults have been introduced. The average function is processed by the software executive system on a hardware platform (Figure 5.14).

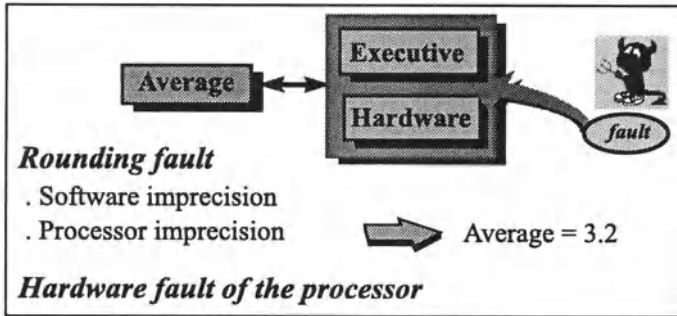


Figure 5.14. Faults due to the execution context

The division of the `Sum` value by the number of elements is necessarily rounded. In order to illustrate roughly the error, we assume that the result is 3.2 instead of 3.25. Such rounding error may have several causes:

- hardware fault of the arithmetic co-processor (one frequently refers to the first version of the Pentium microprocessor which had an ALU bug),
- production fault: the compiler uses a math library having insufficient arithmetic performances, due to natural limitations of the arithmetic processor and the number representation coding (e.g. the IEEE 754 floating point standard).

The activation of such faults, and the error propagation through the hardware/software structure is difficult to analyze because it depends on the real implementation which is generally unknown. Even if the errors seem not to be significant (the precision of one value is not so bad), their cumulative effects can produce significant failures of the mission. Exercise 5.6 illustrates such a situation.

It is very difficult to propose properties allowing floating precision errors to be detected. For this reason, this issue is eliminated, considering two points of view:

- The expected precision is defined during the design and is implemented at programming level. For example, the Ada language features authorize such a definition. Then, studies are carried out on the program in order to analyze the effects of the well-known precision value and the run-time system is considered by providing correct computation.

- The use of floating point representation is forbidden. Only fixed representations are authorized. This restriction is required for the high-dependable systems.

5.5 EXERCISES

Exercise 5.1. Faults of a MOS network

We consider once more the MOS network presented in section 5.2.2.3 (see Figure 5.15).

1. Compare the three logical functions performed by the circuit:
 - for a faultless circuit,
 - for fault *F1* (a MOS is *stuck-OFF*),
 - for fault *F2* (*short* between two lines).

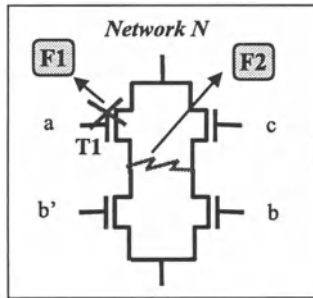


Figure 5.15. Faults of a MOS network

2. The logical function of the faultless circuit is not modified if inputs *b* and *c* are permuted. What is the influence of this permutation on the failure induced by fault *F2*?
3. What is the influence of the ‘stuck-ON’ of transistor *T1* (the transistor is always in an ON state)?

Exercise 5.2. Faults of a full adder

Let us consider the full adder of section 5.4.1. We want to study the influence of functional and hardware faults on the behavior of the circuit, and to determine the resulting failures.

1. Study the following functional fault, noted *F1*, introduced during the design phase: the EXCLUSIVE OR gate has been transformed into an

IDENTITY gate (its output is '1' if and only if the two inputs have the same value).

2. Study the hardware fault, noted $F2$, occurring during the operation: a 'stuck-at 0' noted α in Figure 5.16.
3. Compare the failures provoked by these two faults.

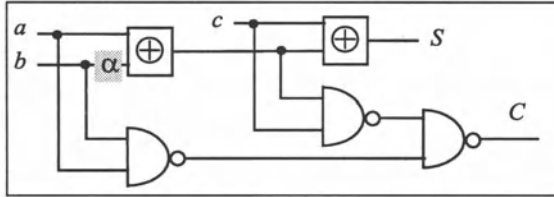


Figure 5.16. Full Adder

Exercise 5.3. Fault models and failures

Consider again the circuit of Example 5.1.

1. Determine by formal analysis the failures provoked by each fault of the single stuck at 0/1 model. Draw and compare the resulting truth tables.
2. Make the same analysis with various functional faults transforming the AND and OR gates.

Exercise 5.4. Faults of a sequential circuit

Figure 5.17 shows a Moore sequential synchronous circuit having one input (x), one output (z), and two internal variables ($y1, y2$) materialized by two synchronous D Flip-Flops.

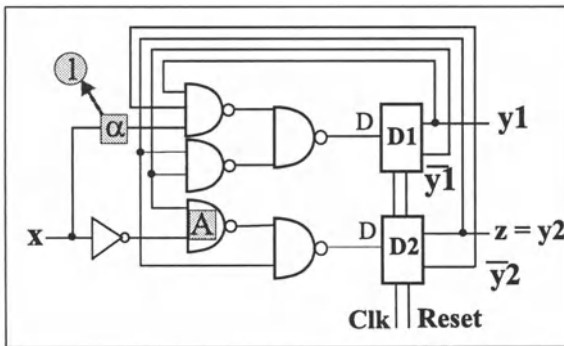


Figure 5.17. Sequential circuit

1. We suppose that a functional fault made during the design of this circuit has led to the transformation of the NAND gate denoted A into a NOR gate. Analyze the new circuit in order to determine all induced failures.
2. Now we consider the hardware 'stuck-at 1' fault of the line noted α in the figure. Analyze the altered circuit to determine all failures.

Exercise 5.5. Software functional faults

This exercise refers to the program computing the average value of a set of numbers, presented in section 5.4.2.

```
function Average (A: in Set) return Element is
Sum: Element:=0.0;
begin
  for I in A'range loop
    Sum:=(A(I) + Sum)/2;
  end loop;
  return Sum;
end Average
```

Figure 5.18. Design faults

1. Consider the two design faults of lines 5 and 7 shown in Figure 5.13. Analyze these faults and determine all the input vectors which provoke failures. Compare the seriousness of these failures.
2. A third fault has led to the following program (Figure 5.18). What are the input activation conditions and their relative failures? What external consequences could result from these failures?

Exercise 5.6. Software technological faults

We want to develop a program which calculates and prints the sum of the float numbers ($1.0 / \text{float}(I)$), for I varying from 1 to a given value N . The following algorithm performs such calculus:

```
procedure Serie (N: in integer) is
Sum : float := 0.0;
begin
  for I in 1..N loop
    Sum := Sum + (1.0 / float (I));
  end loop;
  Print (Sum);
end Serie;
```

Then, execute this program for different increasing values of N . You may conclude that the series $\sum 1/I$, $I = 1, N$ is convergent. This conclusion is mathematically wrong. So, your program fails. Where is the fault?

Chapter 6

Towards the Mastering of Faults and their Effects

The science of dependability is closely linked to the mastering of faults and their internal and external effects. It firstly implies the analysis of destructive mechanisms (problem identification). They were studied in the previous chapters. Then, protection methods (use of means) must be proposed. This aspect is introduced in this chapter. Finally, one must assess the efficiency of the use of these methods on the dependability of the final product. This last aspect (performance evaluation) is exposed in Chapter 7.

6.1 THREE APPROACHES

In Chapter 3, the causes of product failures were classified into several groups:

- *internal functional faults* (or *creation faults*) of specification, design and production,
- *internal technological faults* (or *physical*, or *hardware* faults) of production and operation,
- *external functional faults* (or *perturbations*, or *disturbances*) due to the functional environment,
- *external technological faults* (or *perturbations*, or *disturbances*) due to the non-functional environment.

We have also observed the transformation of faults into internal errors, then into functional failures, and finally their external effects called consequences. We have insisted on the remarkable properties of faults: they

seem to be inevitable, their occurrence follows statistical rules, they can accumulate throughout a life cycle, and they are difficult to master. We identified a destruction process having two steps:

- the occurrence of faults produced at the three first stages of the life cycle considered (specification, design and production), and/or faults due to the product's environment in operation,
- and then, during operation, a transformation into several stages, from faults to errors, then to failures, which finally have external consequences on the application.

Therefore, in order to face these problems, we will use several actions, operating on the fault causes as well as on their effects. Traditionally, the different dependability techniques are distributed according to three complementary approaches illustrated by *Figure 6.1*:

- *fault prevention*,
- *fault removal*,
- *fault tolerance*.

We should note that the fault prevention and fault removal approaches are sometimes regrouped by the term *fault avoidance*.

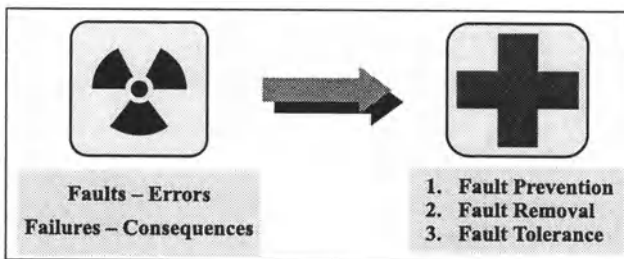


Figure 6.1. Protective mechanisms

The objective of this chapter is to briefly explore these three dependability approaches in sections 6.2 (fault prevention), 6.3 (fault removal), and 6.4 (fault tolerance). Each section aims at defining the goal of the related approach and at giving an overview of the main techniques. Then, we introduce the dependability assessment problems in section 6.5. Fault prevention, fault removal and fault tolerance techniques will be analyzed in the third and fourth parts. Thus, this chapter provides an overview of the means which will be detailed in the following chapters of the book.

6.2 FAULT PREVENTION

The *fault prevention* approach consists in avoiding or reducing the introduction of faults during the specification, design, production, and operation stages, and/or in reducing the occurrence of faults during the product's use. Two complementary action groups limit fault occurrence:

- mastering the stages of the process used to create a product and to use it,
- and acting on the technological means.

The first group of means acts on the faults committed by humans or by the tools they use during specification, design, production and operation. The second group, which is independent of the first one, is dedicated to the faults which are due to the technological degradations of the product, or which are introduced by environmental aggressions.

Fault prevention techniques aim at obtaining and maintaining a product 'without any fault'. However, even if these techniques are efficient, residual faults frequently exist before the operation step or faults appear during this operation step. Hence, this remark justifies the use of *fault removal* and *fault tolerance* techniques.

6.2.1 During the Specification

The initial contract that binds the different partners of a project must firstly express, in a complete and correct manner, all the product's functional and non-functional aspects of the needs that the final product has to satisfy. Any *incompleteness* in the product definition can lead to failures in the final product. Indeed, such a product cannot offer the expected functionality and performance if they have not been expressed by the client or by the product's user. However, all missing specification information does not necessarily imply a future failure; it could represent a degree of freedom for the designer. For example, the specification of a coffee distributor is to deliver the coffee and give back change without defining in which order the two operations need to be carried out. A particular design will make a choice which will be accepted by clients and users.

Moreover, the contractual base has to be understood in the same way by all the partners: the specifications have to be *non-ambiguous*. The ambiguity expresses the fact that precise yet different definitions can be given by the client, the designer and the user, which represents a potential source of failures. The removing of all ambiguity requires the use of a formal specification language, for which the *semantics* (the feature meanings) has to be precisely defined. In practice, the majority of clients and users is not familiar with such languages. Therefore, we will firstly define specification

needs written in a human language by using terms which are clear and precise as much as possible. These terms need to be defined by a glossary.

We should note that the transformation of requirements into formal specifications is not systematically carried out previous to a design. In many industrial projects, informal specifications are the designer's principal source of information. On the contrary, in other cases, formal specifications are requested which are established by a specialized team, after consulting the client and possibly the user. The formal specifications provide therefore a new expression of the requirements. Hence, this activity appears redundant and therefore useless to certain persons. On the contrary, when aiming towards dependability, this rewriting allows the needs to be correctly understood (because they have to be reformulated in another language) and to detect possible incompleteness or ambiguity.

6.2.2 During the Design

Fault prevention is obtained in diverse ways, by acting on the product that is being designed, and/or by acting on the creation process itself. The designer has to apply a 'good' method and apply it 'correctly'.

Firstly, it is necessary to master the design process, from the specifications until the system is designed. The design is a top-down process of several stages, which begins with the *specifications* and ends with a *system* by successive transformations. We should remember that the *system* is an abstract vision of the future product. In order to avoid the appearance of faults during this process, we have to choose transformation models and methods that guarantee that the obtained system conforms to the specifications or is compatible with them. According to the notations used in Chapter 4, we must have $\Sigma = S$ (or $\Sigma \geq S$), that is to say the functioning of the system has to be equivalent to the specifications (or greater: to perform more functions). The choice of models, methods and tools is fundamental.

Secondly, the modeling tools and design process being defined, the way they are used must be mastered. For instance, programming languages are modeling tools used for software design. Their features can be assessed, taking dependability criteria into account. Thus, the 'best' (most suitable) language can be selected. Moreover, a bad-programming style will be at the origin of numerous faults in produced programs.

6.2.3 During the Production

Fault prevention methods during production concern above all the hardware products. These methods imply to master the manufacturing and assembly process of electronic components. The techniques used depend on

the nature of the components. Thus, for integrated circuits, it is necessary to control the parameters of diverse machines, which compose the integration chain and the environment conditions of the rooms that shield this equipment. This particularly concerns dust removal, as dust constitutes an important source of faults. In order to ensure that the production chain functions correctly, several samples of the manufactured components are then subjected to a rigorous control by means of electrical, mechanical, microscopic and chemical tests. This *quality control* allows an improvement of the quality of production.

Where software is concerned, the production phases are in general automated. They use:

- a run-time environment associated with a programming language composed of a compiler, a linker and an executive (real-time kernel, input/output library, etc.),
- software components developed by other companies and which are often called *COTS (Components Off The Shelf)*.

The dependability of the executable application obviously depends largely on the dependability of previous elements. The need for dependability therefore has an important impact on their choice and their implementation. For example, the choice of an execution environment associated with a programming language has to prove that it has successfully passed standard tests, if they exist. If these standard tests do not exist, the developer has to write specific local tests and to apply them to the execution environment to be evaluated. In the same way, the use of already developed software and hardware components (called *reuse*) does not prevent their analysis in terms of dependability before their integration in the product.

6.2.4 During the Operation

The creation problems having supposedly been resolved, the failures that appear during the useful life arise from a bad utilization of the product or from perturbations acting on the technology and due to the non-functional environment.

The prevention of numerous external faults caused by the user can be obtained by simplifying and making clear the use of the product in its context, e.g. easy to understand user manual, user friendly interfaces, on-line contextual help. This is the typical case of popular convivial software of *Macintosh's Apple* that we find today in many *UNIX* or *Windows* systems.

A second approach consists in designing protection means which prevents the occurrence of errors, even if they are due to incorrect use of the product. For instance, if a user has to select an action on a computer, the first

approach consists in displaying a menu and in getting the keypressed character indicating the chosen action. However, a wrong character may be provided. A second solution displays a list of buttons associated with the offered actions. Therefore, the user cannot give unknown orders. A second example concerns the use of connectors, which cannot be incorrectly plugged in. For instance, different categories of connectors are not compatible together: electrical power, phone, printer, screen, modem, etc.

Concerning the faults associated with technological implementation means and the non-functional environment, we can distinguish hardware and software technologies.

Hardware faults appear during the operation phase due to component ageing or external aggressions. Their occurrence follows statistical models of reliability. The factors that determine the occurrence of these breakdowns depend on the technology used and the environment (temperature, vibrations, shocks, radiation, etc.). We can first of all reduce the occurrence of faults by choosing design and manufacturing techniques which lead to *high reliability* products: choice of components and mounting and assembly techniques which reduce the probability of faults appearing. The electronic components (ICs, PCBs, etc.) can also be submitted to *burn-in* operations where they are used at high temperature for periods which can be longer than 24 hours in order to provoke faults of weak elements; only the components which survived are put on the market. Thus, the infant mortality rate of the used systems is drastically reduced.

Then, the faults induced by the environment can be prevented:

- by protection techniques, for example electromagnetic shielding, thermal isolation, etc.,
- and/or active observation techniques of the environment's parameters, for example an on-line observation of a microprocessor's temperature in order to detect anomalies, such as a ventilator breakdown, which could lead later to component faults.

Software technology does not age in the same way and is also not subjected to external aggressions. However, phenomena having similar effects can happen. The software's operational life can sometimes be relatively long (twenty to fifty years for a software used in an aircraft), and the execution means can vary during time. For example, changing a processor for a newer circuit which is quicker and has new functionality, can lead to a modification of the execution environment. These evolutions can have not only an impact on performance, but also on the behavior of software which could provoke failures. For example, increasing the frequency of a processor's clock or the optimization of a code generated by a new compiler can reduce the processing time and induce modifications of

the sequencing of executed tasks and therefore the product's behavior. Where the programming technology is concerned, these faults can be avoided if the language used has precise semantics. Thus, the diverse execution environments of this language will produce applications which have identical behavior.

6.3 FAULT REMOVAL

6.3.1 General Notions

Fault removal aims at detecting and eliminating faults present at the end of each specification, design and production stage, as well as faults appearing during use. The fault suppression techniques are different for each one of these stages; however, three aspects are explicitly or implicitly taken into consideration:

- **fault detection** which reveals the presence of faults,
- **fault localization** (or **fault diagnosis** or **fault isolation**) which identifies the faults present,
- **fault correction** and/or **repair** operation which deletes these faults when the systems allows it (such as with **repairable products**).

Independently of these notions of detection - localization - correction, we identify two distinct groups of fault removal techniques:

- **static analysis** techniques,
- **dynamic analysis** techniques which are also known as **test** techniques.

Static analysis is carried out 'without execution' of the analyzed model. This integrates:

- **formal proof techniques** by equivalence between the model which is treated by the stage and the model obtained at the end of the stage, or by researching particular properties of the obtained model, etc.,
- **review techniques**, greatly used in the software domain, which consists of an analysis of the system by human experts.

On the contrary, the **dynamic analysis** or **test** is carried out by executing the model analyzed. We test a system or a product by subjecting it to stimuli from the outside. More precisely, we apply a sequence of data on the inputs and then we observe the behavior on the outputs (sometimes, we observe also some internal signals or variables). This is typically an **experiment**.

In Chapter 2 we remarked that the transformation of requirements into a product is carried out by a succession of phases. An elementary phase transforms the model of the level i to the model of level $i+1$ (see *Figure 6.2*). This process is therefore said to be *top-down*. The detection of faults could consist in examining this transformation to see if the model obtained at $i+1$ level is in accordance with the initial model at level i . Whatever the technique used, we are speaking here of *verification* which is symbolized by a *bottom-up process* symbolized by an arrow in *Figure 6.2*.

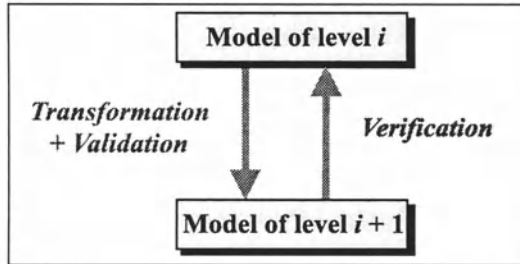


Figure 6.2. Fault removal during one transformation

The faults introduced stem from a bad transformation of the initial model. We are therefore naturally led to question the quality of the transformation method chosen. If it is possible to establish that this method is bad, we may assume that using a better method could have prevented faults. This approach is different from that of verification. We do not seek to detect faults but highlight the high risk of faults being introduced. For example, a program review makes obvious that certain ‘programming rules’ have not been respected. The means relevant to this process are qualified as *validation* techniques. A *top-down process* symbolized by an arrow in *Figure 6.2* symbolizes this validation process.

We should note that the two words *verification* and *validation* unfortunately have different meanings according to the domain considered.

Fault removal during the development phases of an industrial product is very important but expensive in terms of human and technical means and also time consumption. Faults must be detected and removed as soon as possible according to the adage: “finding faults early in the design cycle directly impacts the development cost and schedule”. Year after year, the design of Integrated Circuits is improved, and their fault rate decreases; however, as the complexity (number of transistors) of these chips increases in the same time, the difficulty to pinpoint the residual faults increases: *Figure 6.3* illustrates this historical evolution with a symbolic ‘Fault - Test complexity’ diagram.

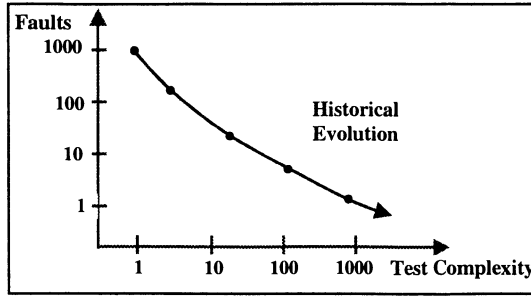


Figure 6.3. Fault removal rate

The following sub-sections introduce the objectives and principles of static and dynamic analysis methods used during the stages of a life cycle. These techniques are discussed and developed in the third part of this book.

6.3.2 During Specification and Design

The fault suppression methods applied during the specification and design stages concern the functional *creation faults*.

6.3.2.1 Static Analysis

Detection

The modeling means used to specify or design a system offer *features*. For example, the ‘finite state machine’ model is based on two *features*: the *state* and the *transition*; the features of a programming language are described in its reference manual. These constructions possess two essential characteristics: the power of expression and the detection capacity.

The modeling tool features have essentially been introduced to facilitate the expression of models for the levels where they are used. For example, during the programming stage (which is part of software’s design), the feature of the loop `for` was introduced to easily express repetitive treatments. In electronics, the ‘register’ concept was created to express the need to store data. Where the design of real-time systems is concerned, the notion of ‘task’ was introduced to easily handle asynchronous events.

The features were then created to make the detection of modeling errors easier. For example, in the programming domain, the notion of ‘type’ allows the compiler to detect situations where an expression of a type is assigned to a variable of a different type. In the same manner, we can quote the example of the Petri nets whose analysis reveals potentially reachable non-desirable states or deadlock situations. This second characteristic of these features was introduced more recently, whereas it is essential when dependability is an

important criterion of a product. Thus, the choice of modeling means has to be made taking into account these possibilities of fault detection as much as their power of expression.

Once a modeling tool is chosen, two classes of errors can be detected.

- The *generic errors* associated with the considered model means and not to the characteristics of a particular modeled system. For example, a blocking of a system's behavior described by a Petri net can be detected, whatever the functionality of the system is.
- The *specific errors* of each modeled system. An example has already been quoted of the reachability of particular state of a graph that is considered undesirable. Thus, for a rail regulation system of a level crossing, the state 'the barrier is open' AND 'the train is passing' constitutes an error that we would like to detect.

It is really errors and not faults, which are made obvious since they are about undesirable states and not structural elements which have led to these states. For example, we do not know (for the moment) the piece of program or circuit which provoked the opening and staying 'open' of the barrier when the train passed.

Localization

The localization of a fault by a model's static analysis will be easier if the used modeling tool proposes features which favor the expression of specific erroneous behavior. Indeed, the analysis could then detect an error close to the original fault and not later, through errors induced by contamination mechanisms (side effects). For example, if the model used allows the detection of a blocked system, the designer would want without doubt to know the functioning sequences which led to this undesirable situation. The localization of the fault is therefore made easier. Here again, the choice of a modeling means has to be taken by analyzing the facilities offered by the model to reveal faults which are at the origin of the detected errors.

Correction

Once again, the characteristics of the modeling means and the way which this means is used have an important impact on the facility and efficiency to correct the localized fault. Consider, for example, a designed system made of loosely coupled components, that is to say having few interactions and whose actual interactions are explicit. Therefore, the correction will be localized in a part containing the erroneous component. On the contrary, a strong coupling between components makes the correction difficult, due to constraining relationships which links them; it is therefore necessary to modify several components in order to avoid the error occurrence. The risk

is then to introduce other faults during this correction. For example, if a program contains numerous global variables (shared by several procedures), the modification of one part of the program (a procedure) risks having indirect consequences on other parts of the program.

When a fault is introduced during the specification and design process, we can ask ourselves if this results from an ‘unfortunate accident’ (absent-mindedness, etc.) or if it originates from a fault in the development process itself. This question is important, as in the second case several similar faults probably exist. To correct these faults one after the other is inefficient; therefore it is necessary to diagnose (localize) the fault in the development process and correct or improve the development process. This case corresponds to a *validation* process approach and leads to the introduction of new techniques belonging to fault prevention. For example, if we state that numerous faults found in a software product are due to an unclear distinction between the variables which are ‘local’ and the variables which are ‘global’, we would introduce a guide defining a programming ‘style’ which imposes rules (or constraints) on the choice of variable identifiers.

6.3.2.2 Dynamic Analysis

Let us remind that dynamic analysis, often called *test*, implies the execution of a model. This supposes the existence of a formal semantics of this model as well as associated execution means, and sometimes an external model of the product’s behavior. We do not consider here the particular case obtained at the end of the design where we dispose of a physical executable product mock-up. This case will be examined during the production and operation stages.

Detection

Fundamental fault detection principles of dynamic analysis consists in:

- applying a sequence of input data to the system model in order to transform the faults into errors,
- and by detecting erroneous states by any mechanism which observes generated states, including outputs.

The model therefore has to allow the expression of correct states and those which are considered as incorrect. The execution of this model has also got to include the comparison of the system’s real states, whether with those expected (supposedly correct), or whether with the non-desired states (supposedly incorrect). In the software domain, the definition of ‘constrained types’ has been introduced with this in mind. Consider as an example ‘`subtype Size is Integer range 28 .. 48;`’. At the program’s execution, the value assigned to a variable of this type has to

belong to this type (the interval [28,48] for our example); the violation of this *assertion* detects the error.

Another example, typical of dynamic analysis used for the detection of errors, is the *test by simulation*: a functioning sequence is applied to the product model and the outputs are then compared with the expected predefined outputs. Such an analysis, which transforms faults into failures, is applicable in numerous situations because it makes use of an external knowledge of the system, independently from its structure.

The dynamic analysis confronts us with two problems: the *controllability* and *observability* of faults. It is necessary to find a functioning sequence which activates the faults into errors: this expresses the *controllability* notion. However, this error creation is not sufficient for the detection; we also have to be able to observe these incorrect states (errors): this induces the *observability* notion. If we dispose of an 'encapsulated' product (only the inputs/outputs are accessible), the dynamic analysis is totally carried out externally: controllability and observability are minimal. On the contrary, in the case of a software written in a language which has an 'exception mechanism', a certain number of abnormal states are signaled; thus, the violation of a constraint such as the one associated with a variable whose type 'Size' was previously defined could, for example, provoke an error message and stop the program execution. Then, the observability is high.

In conclusion, the detection of faults by the execution of a modeled system reveals two problems:

- How can the system be brought into an erroneous state?
- How can we perceive that the system has reached such a state?

We should note that the fault models expressing the faults that can affect the specification and design stages are generally difficult to establish precisely. We often apply the analysis of erroneous behavior by detecting the errors and not the faults which are at the origin of these errors.

Localization

Even if the existence of an error is signaled during the execution of a modeled system, the localization of the fault which is at the origin can turn out to be difficult. Indeed, we have to go back to the system structure, that is to say to go from the failure or error to the original fault. It is clear that this work is facilitated if the distance between the place where the error was signaled and the place where the fault occurred is reduced. Consider an example from the software domain. Let an assignment statement place the result of a function in a variable *P* of Size type:

```
P := function(V1, V2, V3);
```

Suppose that the execution of this statement signals a violation of the

constraint associated with the variables of type *Size*. If the input variables, *V1*, *V2*, *V3* are also constrained by types and if no type violation has been detected in these variables, this suggests that the fault could be located in the function and not backwards. However, this is not a certainty. For instance, the function can be correct but should have been called at the wrong place.

Thus, localization is made easier by using executable modeling means whose constructions have numerous and precise detection mechanisms.

Correction

The remarks made regarding static analysis could also be applied to dynamic analysis.

6.3.3 During the Production

As the specification and design faults have now supposedly been prevented or eliminated, we now look at the faults introduced by the production process. Stemming from this process step, the product therefore exists and we can subject it to dynamic analysis tests. The investigation means are therefore generally applied through the normal inputs/outputs: controllability and observability are a priori limited. However, where the electronic aspects of production are concerned, we dispose of relatively precise models of the faults which can affect the product, whether at the production means level (manufacturing and assembly machines, etc.) or at the level of the product itself (for example breakdowns could affect a component). Finally, the production imposes certain constraints on the test techniques: in particular the duration of the tests should not slow down excessively the rhythm of production.

Detection

A product being by definition executable, we employ mainly dynamic analysis methods: we apply a sequence of input values to the product, and we observe the output values produced which are then compared with pre-established values or with the output values of a 'standard' product which is supposedly perfect. Static analysis methods also exist, such as the visual inspection of a printed circuit board to detect insertion or welding errors of electronic components.

The detection of production faults is known as *production testing*. Whatever the methods used, this test is characterized by a short execution period, contrary to *design testing*. Indeed, if the cost of design testing is spread out over all the manufactured products, the cost of production testing has to be added to that of each product. Therefore, it is necessary to detect as quickly as possible the largest number of faults belonging to a technological

fault model. This wager implies a compromise between the demands for correctness of the manufactured products and the costs and delays in the manufacturing.

Localization

The localization or diagnostic or isolation of eventual faults is generally made difficult by the inaccessibility to the internal elements. This operation requires a stricter and longer analysis of the product which is only accepted by the manufacturer if the detected failing components are aiming towards an improvement in productivity. In the domain of electronic circuit manufacturing, we proceed to a *quality control* on component samples. Therefore, we look for the causes of such failures:

- in the product (for example a breakdown due to a component overheating),
- in the manufacturing process (for example the temperature of welding equipment is badly regulated).

Correction

When the localization analysis reveals a flaw in the production process, we have to act on the process in order to correct the flaw. In certain cases, we can repair the detected failing product by changing a component, a connector or by re-establishing a broken line, etc.; the product is therefore said to be *repairable*. In the opposite case where the product is *non-repairable*, its failure constitutes a financial loss which reduces the yield of the production since the product is excluded from commercialization.

6.3.4 During the Operation

During operation, we meet functional and/or technological and/or environmental faults. The operation test also known as a *maintenance test* corresponds to the same type of problems as the production test. However, it is often more precise because it is longer, and is not subjected to the temporal constraints of production. In the case of repairable systems, we look to the diagnosis of the fault present in order to correct it rapidly.

The set of manufacturing or maintenance test techniques constitutes the *off-line testing*. This test is called as such because the normal functioning of the product has to be stopped in order to test it: for example, taking a car to the garage for technical service.

On the contrary, we qualify the mechanisms which carry out self-tests on the product during its functioning as *on-line testing*. For example, a light turns on in a car if there is not enough gas or oil, or if the engine temperature

is too high. More complex procedures are also applied at run-time on control systems embedded in all recently produced cars.

In general, the detection or localization tests of average or high complexity systems are very difficult to establish, and the sequences obtained are very long. This has repercussions on the cost and duration of the test during the life cycle. A certain number of solutions to introduce as early as the design stage are proposed in order to make the final product *easily testable*. In particular, error detection and fault diagnosis are facilitated by using *instrumentation* techniques. Observation means are put into place, which observe on-line anomalies and record typical variables; hence, the final diagnostic carried out during the maintenance is made easier. This is also referred to as *monitoring*.

6.4 FAULT TOLERANCE

The two approaches, fault prevention and fault removal, go towards the development of dependable applications by providing:

- means which facilitate the expression of specification and design modeling and transformation, hence reducing fault introduction or occurrence by acting on the product's creation process,
- means which permit fault detection, and means which avoids their propagation along the development stages.

The *fault tolerance* strategy is different to those of the two other approaches: it involves acting on the effects and not on the causes! We begin with the realistic hypothesis that, despite the previous methods of fault prevention and removal, the product used remains affected by the residual faults arising from the design and manufacturing stages. Independently from these residual faults, technological faults will probably appear during the use of the product. Tolerance to faults is generally based on *redundancy* techniques (for example by duplicating components) which act during the product's use, although they have been defined and implemented during the specification, design and production.

It should be noted that the aim of fault tolerance is not to correct the cause of the error (the fault) but to prevent the appearance of a failure. Thus, if a program contains a design fault, the tolerance mechanisms do not modify the faulty statements. In the same way, if an integrated circuit is affected by a breakdown, the mission should carry on functioning despite this breakdown.

In order to make our presentation clearer, we distinguish three significant classes of fault tolerance techniques:

- the *failure prevention* which proceed by *error masking*,
- the *detection and correction of errors* at their occurrence,
- the handling of faults and errors to prevent the occurrence of ‘dangerous failures’: here we are speaking of *fail-safe systems*.

Briefly introduced and commented on in the following sub-sections, these classes will be discussed in the fourth part of the book.

6.4.1 Failure Prevention by Masking

The *error masking* techniques involve redundant devices which inhibit the effects of faults, thus preventing the appearance of failures. The most popular example is the *TMR (Triple Modular Redundancy)* which corresponds to a redundancy by ‘triplication’ (also called *triplex*) of the hardware and/or software modules. The outputs of the global system are obtained by a majority vote of the outputs of the duplicated modules. Thus, an erroneous result of one of these modules has no effect on the final output. This principle of redundancy is also used to secure the wheels of a lorry: by placing several wheels in parallel instead of one single wheel allows the effect of a burst tire to be masked.

The redundancy used according to this first approach is qualified as *passive redundancy* because it does not require a mechanism to detect the error or modify the product’s action in order to prolong its mission. The corollary of this principle is that faults can hardly be detected externally.

This historic approach was used in the first space projects requiring high dependability because it is simple. It is however abandoned today, in favor of the second approach by *active redundancy*, which is more flexible and efficient. This second approach is described in the following section.

6.4.2 Error Detection and Correction

Contrary to the previous approach, the techniques based on *error detection and correction* mechanisms necessitate the explicit detection of errors produced by the faults, then the use of means allowing the correction of these errors. An example of this in daily life is the spare wheel in a car: the driver has to sta

The redundancy implied in this approach is qualified as *active redundancy*, as it requires an explicit activity to detect the appearance of errors and to handle them.

An illustration of this use of active redundancy is provided by the *error detecting and correcting codes* presented in Chapter 15, and used for example to code data stored in CD-ROM devices.

This approach calls for three groups of complementary techniques:

- *self-testing* or *on-line testing*,
- *fault contention* or *error confinement*,
- *reconfiguration*.

With the *self-testing systems* (or *on-line testing systems*), the presence of a fault is detected by a mechanism which signals the occurrence of an error.

We can then, with the second group of techniques, called *fault contention* or *error confinement*, prevent the error from reaching other modules or functions of the product.

Finally, with the third group of techniques, called *reconfiguration* techniques, the product is adapted to continue its mission, as long as its resources permit this, with or without a degradation of its performance. For example, the installation of a car's spare wheel only allows one flat tire to be replaced; the spare therefore constitutes a 'resource': a single puncture does not therefore imply reduced vehicle performance (if we do not count the halt of the vehicle and the time spent to change the tire.) On the contrary, if this spare wheel is a 'light' type (that is to say a wheel of reduced width), the vehicle whose tire has been changed will only be able to drive at reduced speed: therefore this implies a degradation of performance.

6.4.3 Fail-Safe Techniques

The techniques which prevent dangerous failures concentrate essentially on the product's *safety* criteria, that is to say that they aim at preventing the appearance of failures which have dangerous or catastrophic external effects. We are speaking here of *fail-safe systems*. Take for example the electronic regulator of traffic lights on a crossroad: the failure which provokes the action 'green - green' in both directions is reputed to be dangerous. We should therefore design this regulator in such a way that the probability of occurrence of this failure is very low (below an acceptable level).

We should note that the prevention of dangerous failures is independent of error detection and correction techniques of the previous approach. However, we will see that the methods used are often close to the self-testing methods.

6.4.4 Resulting Fault Tolerance Classes

The objective of fault tolerance is clearly to prevent failures. For pedagogical reasons, we are going to constitute three groups of techniques classified by their increasing complexity:

- *self-testing systems* which ensure the simple detection of errors during the product's functioning and often constitutes the first stage towards fault tolerance,
- the *fail-safe systems* which prevent failures considered as dangerous,
- *fault-tolerant systems*; this fault tolerance is either *passive* by masking, or *active* by self-testing and error correction/reconfiguration.

6.5 DEPENDABILITY MEANS AND ASSESSMENT

Figure 6.4 integrates the different aspects of dependability: the impairments, the handling means, as well as the techniques allowing the evaluation of dependability. It is indeed necessary to be able to measure the impact of the diverse techniques used on the dependability grade of the final product. These measurements are introduced in the next chapter. The *dependability assurance* is the set of scheduled and systematic actions which are taken in order to guarantee that the final product satisfies the dependability requirements.

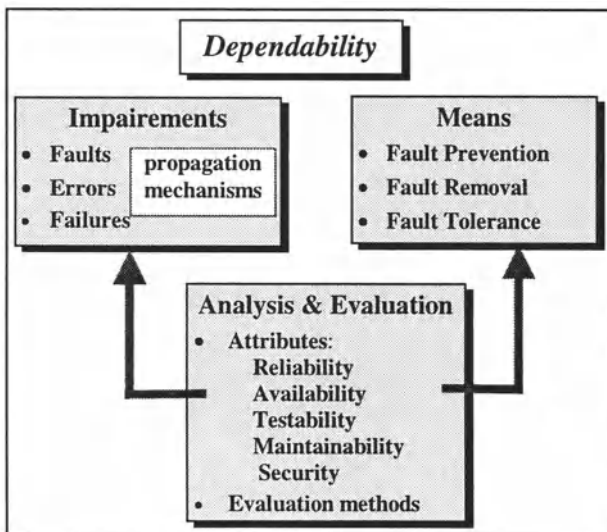


Figure 6.4. Dependability summary

Figure 6.5 illustrates the techniques which stem from the three approaches, fault avoidance, fault removal and fault tolerance, throughout the product's life cycle. On the left, the problems are symbolized, that is to say the types of faults and their cumulative incidence on the stages of life cycle: F_s (specification faults), F_d (design faults), F_p (production faults) and F_o (operation faults). On the right are represented the solutions used at each stage to reduce or suppress faults and their effects.

It should be noted that the prevention techniques associated with a stage of the life cycle concern uniquely the faults which can appear during this stage; the removal techniques treat not only the faults introduced during the stage in process, but also faults stemming from previous stages. Fault tolerance mechanisms act on the product during its use, but their implementation is associated with the product's development stages (specification and design).

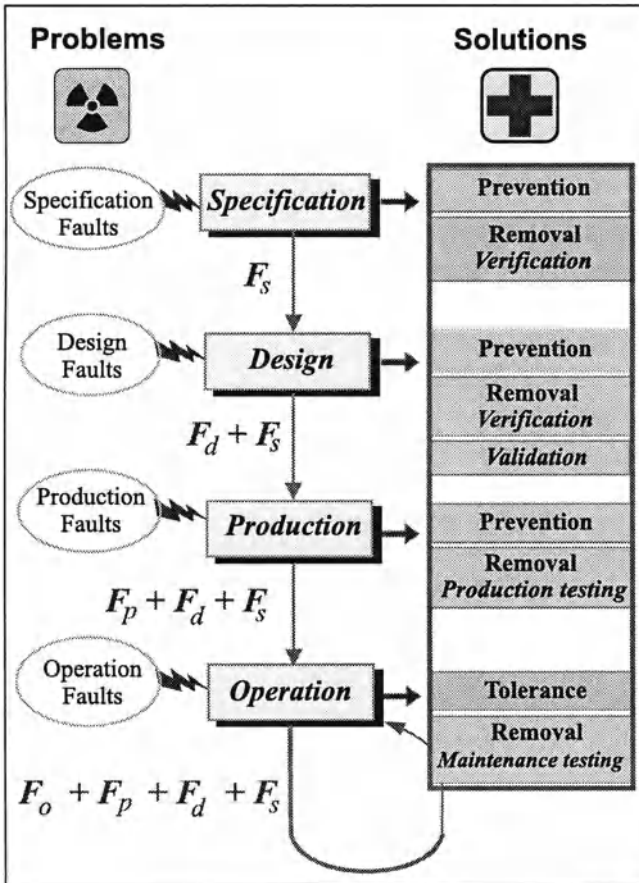


Figure 6.5. Fault mastering

6.6 CONCLUSION

The aim of this chapter was to provide the reader with an overview of the different aspects of dependability science. Parts three and four of the book develop the fault prevention, fault removal and fault tolerance techniques introduced in this chapter. Part three deals with fault avoidance means (fault prevention and fault removal). We separate the functional faults from the technological faults, as the related techniques are often quite different. The very important class of fault removal techniques related to technological faults is studied with more details in three chapters of this part. Part four is dedicated to the fault tolerance means: on-line detection and recovery of errors during the operation stage.

Before analyzing these techniques, we will explain in Chapter 7 the *dependability assessment* issues, and we will introduce in Chapter 8 the basic notions of *redundancy*. Dependability assessment means are necessary to evaluate the various techniques proposed to develop dependable products. Redundancy is a fundamental concept of most of the protective means.

Chapter 7

Dependability Assessment

7.1 QUANTITATIVE AND QUALITATIVE ASSESSMENT

Dependability has been defined as ‘*a property such that reliance can justifiably be placed on the service delivered by a product*’ in its utilization context. Such a definition containing the word ‘justified’ implies means, which allow the evaluation or measuring of the reliance. The numerous assessment approaches which exist are typically classified into two groups. The first one, called *quantitative dependability assessment*, consists in defining dependability measurements and techniques to obtain the values of these measurements. The second one, called *qualitative dependability assessment*, is based on dreaded events and techniques to evaluate their effects and their potentiality. These events are faults, errors, failures and their consequences. Naturally, some of these techniques can be used for quantitative as well as for qualitative assessment.

7.1.1 Quantitative Assessment

To assess the dependability of a product in a quantitative way, several *attributes* can be defined. They depend on the meaning associated with the term *reliance*. These attributes define *probabilistic values*, as the occurrence of a failure is generally not certain. This is due to two causes:

- The fault is due to non-deterministic phenomena affecting the product or induced by the environment; for instance, the date of the arrival of a heavy ion is unknown.

- The fault is due to the development process, but its localization in the product is not defined, otherwise it would have been removed. We only know probabilistic values of these creation faults.

Attributes can be assessed three times throughout the product's life cycle.

- First of all, the attribute's expected values are defined at the beginning of a project. These *specification assessment values* are associated with the specifications of the future product. They are expressed in terms of acceptable *probability* ranges for a given mission. For example, we demand that a new model of a particular light bulb has a probability greater than 0,999 to function correctly during 2000 hours. This demand for reliability must be integrated in the requirements expressed during the product's specification.
- Techniques are then used in order to estimate the *forecasting values* of the attributes during the design. This allows justifying the design and technology choices made during the development stages. For example, the use of `goto` in a program makes the verification of its behavior more complex, and consequently, increases the probability of a program's failure. In the hardware domain, the choice of magnetic technology to store data can lead to insufficient reliability for an embedded product because of space rays.
- Finally, the attribute's values are measured in the operational phase. We then obtain the *exploitation values*. For example, the use of thousands of light bulbs or the use of a program by thousands of users allows the evaluation of the average operation duration without failure.

The evaluation of forecasting as well as operational dependability is generally difficult and does not provide any certitude about a particular product, but only statistical information about the behavior of a population of products. This evaluation is based on several types of deterministic or probabilistic models (*Petri nets, Markov chains, etc.*).

The forecasting evaluation is deduced from knowledge about the product's structure and components. This sometimes uses interpolation and extrapolation techniques to take the final product and its environment into account. Moreover, several candidate architectures could be compared before any design choice. For example, the probability of a rupture of a generator's vapor tube in a nuclear site is $4,8 \cdot 10^{-2}$ during one year. However, the probability that this event leads to a fusion at the heart of the nuclear site is only $1,5 \cdot 10^{-7}$. This smaller probability is due to the setting up of protection mechanisms, from the design stage, which handle the faults.

On the contrary, operational evaluation is based on experimental measurements carried out on representative samples of the population of

products studied; the results are then treated by mathematical tools. This dependability evaluation aims at checking the dependability of real production; in numerous cases, this evaluation is performed relatively to previous products.

The reliability that is desired for the examples of light bulbs or nuclear sites is not the only criteria used to assess dependability. Several other dependability attributes exist. They are normalized by diverse national and international organisms and professional groups, such as the IEEE (Institute of Electrical & Electronics Engineers), the MIL-STD standards of the US Air Force, the ANSI (American National Standards Institute), the IEC (International Electrotechnical Commission), the ISO (International Standards Organization), the (European standard organization), the BS (British Standard) and AFNOR (French normalization organism), etc.

In sections 7.2 to 7.7 we define the following 6 quantitative attributes: *Reliability*, *Testability*, *Maintainability*, *Availability*, *Safety*, and *Security*. These attributes are then compared in section 7.8. Some evaluation tools are introduced in section 7.9: the *fault simulation* approach, the *reliability block diagrams* and the *non-deterministic state graph* approach.

7.1.2 Qualitative Assessment

The qualitative assessment approaches aim at examining *dreaded events* and evaluating their potentiality and their effects. The studied events are faults, errors or failures. The methods are distributed into two classes:

- The *deductive approach* consists in deducing failures from faults or errors (dreaded events). The considered faults or errors often come from previous quantitative studies.
- The *inductive approach* considers potential failures (dreaded events) and establishes which faults or errors can be at their origin.

As for quantitative assessment, the qualitative assessment methods do not handle occurred events but potential ones, whose occurrence must be avoided. In section 7.10, the inductive method called *Failure Mode and Effect Analysis* is presented. A popular deductive method, the *Fault Tree Method* (or *event tree method*), is introduced in section 7.11.

7.1.3 Synthesis

The various qualitative or quantitative assessment methods assume numerous hypotheses, and they are based on the use of models and analysis and measurement means. Each one of these methods provides a particular assessment of the reliance that can be justifiably placed on the services

delivered by the products. Therefore, these methods must be used jointly. *Figure 7.1* illustrates the dependability assessment challenge.

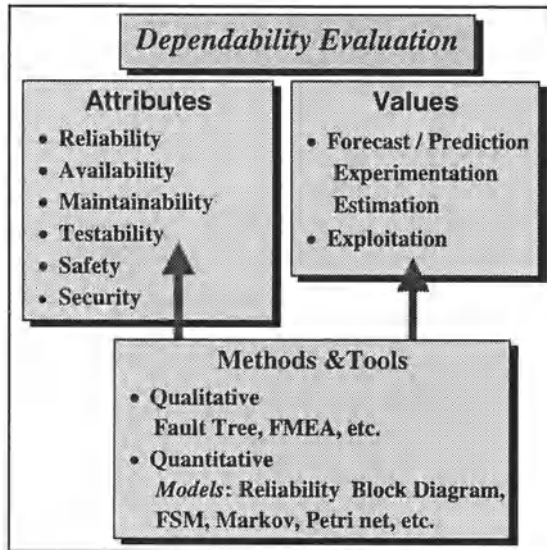


Figure 7.1. Dependability evaluation

As shown in *Figure 7.2*, these techniques can be classified according to the nature of the basic models (events such as faults, failures and their consequences, system with functional and structural knowledge, and physical such as a prototype on which experiments are applied), and the treatment which is applied (probabilistic or statistic). Some of these methods are explained in the following sections.

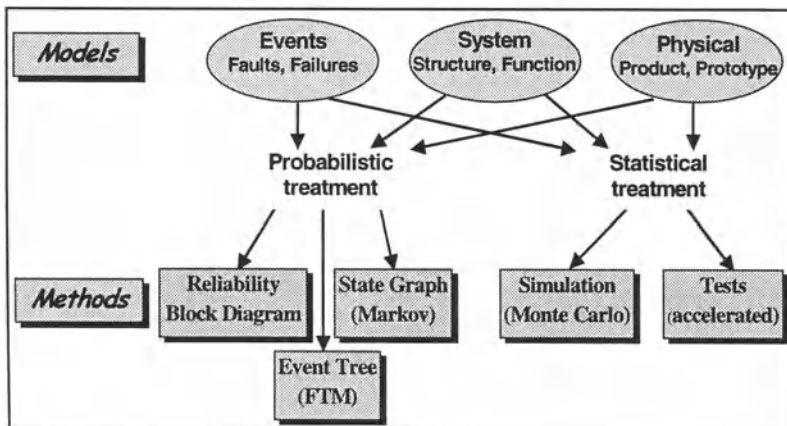


Figure 7.2. Dependability assessment methods

7.2 RELIABILITY

7.2.1 General Characteristics of the Reliability of Electronic Systems

Reliability has initially been defined for physical systems which are submitted to the law of increasing entropy which postulates that these systems have a tendency to degrade throughout time. Reliability is an attribute of dependability with regard to the *continuity of the delivered service*. The international standard IEC85 defines *reliability* as:

the aptitude of a device (product) to accomplish a required function in given conditions, and for a given interval of time.

In a quantified way:

reliability is a function of time which expresses the conditional probability that the system has survived in a specified environment till the time t , given that it was operational at time 0.

A product's reliability is a function which does not increase with time. This decreasing tendency is due to the subjacent phenomenon of the degradation of electronic devices. On the contrary, in the case of software technology, this function stays constant, due to the absence of ageing phenomena (not considering the maintenance operations).

Several statistical reliability models exist. Their pertinence depends on the technology domain being considered. In electronics, the breakdowns are cataleptic (that is to say they appear abruptly, without an external warning sign), and the wearing out phenomenon, classic in mechanics, are considered as minor. The degradation phenomena lead to breakdowns, which are generally represented by models which depend on the environment's parameters such as temperature, radiation, vibrations, etc. The temperature is the dominant parameter: the reliability decreases according to Arrhenius or Eyring degradation models of physico-chemical processes, applicable in electronics.

Reliability studies call for:

- practical mortality *experiments* on samples representative of the analyzed product's population,
- *mathematical techniques* of judgment on samples to deduce quantifiers (reliability parameters) applicable to the whole population.

The first group of studies consists in noting the number of failures appearing during time on the batch of products tested. Following this observation, *statistical description* tools allow to draw reliability curves. Then, *statistical*

mathematical tools allow *estimators* to be deduced such as, for example, the mean lifetime of a circuit, or the failure rate. These tools also allow hypothesis and likelihood tests to be carried out which measure the confidence that can be placed in these quantitative statistical results.

Different types of *reliability tests* are carried out on the populations of components to perform *reliability evaluation*:

- *curtailed tests* whose duration is fixed a priori,
- *censored tests* which stop when a given number of faults is reached,
- *progressive tests* whose decision to stop depends on the results obtained,
- *progressive curtailed tests* which are identical to the progressive tests with a maximum duration constraint,
- *step stress tests* which provoke a progressive acceleration of the degradation mechanisms, in general by increasing the temperature which permits an *accelerated test*.

7.2.2 Reliability Models

7.2.2.1 Exponential Law

Reliability is analyzed by *reliability models* which are mathematical functions of time. The *exponential law* is the simplest of these laws. It expresses the probability of survival by an exponential function which decreases in relation to time (see *Figure 7.3*):

$$R(t) = e^{-\lambda t},$$

where λ is the *failure rate* expressing the probability of failures occurring per hour, for example 10^{-6} failure per hour.

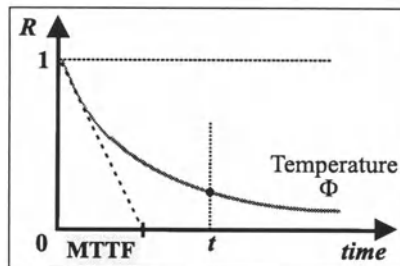


Figure 7.3. The exponential law

λ is generally considered to be constant throughout time. For example, a computing system with a CPU, a Memory, and an I/O unit has a failure rate

of 10^{-5} failure / h. This property correspond to faults following homogeneous *Poisson process* (the average number of faults by time unit is constant).

The preceding exponential law has to be defined for a given temperature, for example 18°C (environment's parameters). Special abacuses (Henry's curves) permit the deduction of the failure rate of an electronic component of a given technology at any temperature belonging to a given range.

This $R(t)$ law is often associated with simple *estimators* called:

- **MTTF: Mean Time To Failure** (also called **MTFF, Mean time To First Failure**) for non-repairable products, for example a mission which terminates as soon as a breakdown happens.
- **MTBF: Mean Time Between Failures**, for repairable products, for example the product which has broken down is repaired and put back into service.

If an exponential law has a constant failure rate, the average value of this function is:

$$MTBF \text{ (or MTFF) } = 1/\lambda.$$

This is expressed in exponential values of 10 hours: for example 10^6 H.

This estimator is often used as commercial arguments in a misleading way: if someone declares that a product has a MTTF of 10^6 , this does not mean that it will survive for this duration! Indeed, in Exercise 7.2 we will see that, at the end of a period of time equal to MTTF, the product in fact only has a survival probability inferior to 37% ($1/e = 36,7879\%$). This remark justifies the observation about the worth of a product only based on its MTBF/MTTF. Many specifications of high dependability projects (for example in the aerospace domain) demand a probability of survival at the end of the mission much higher than the reliability at the MTBF/MTTF value: for example, $R = 0.9999$ at the end of 10^5 hours of mission!

7.2.2.2 Weibull Law

The *Weibull* model is the most interesting reliability law because of its flexibility in describing a number of failure patterns concerning electronics. A simplified version with two coefficients, η and β , is given by the relation:

$$R(t) = e^{-(t/\eta)^\beta}$$

When $\beta = 1$, this law reduces to the exponential law with $\eta = 1/\lambda$.

In the following part of this book, we will only consider the *exponential law*, which is the simplest and the most frequently used law for electronic systems.

Other more precise laws, such as the *Weibull's law*, are unfortunately more complex to understand and manipulate.

7.2.3 Failure Rate Estimation

Failure rate estimations are generally determined from survival tests applied to significant large samples of components. The duration of these tests is short as compared with a product's normal life cycle; this reduction of the test duration is due to an increase in the environment's temperature. Then, by using an acceleration factor we can perform a conversion from high temperature stress test to equivalent nominal operating system temperature. Thus, the failure rate of the circuits is deduced. These experiments are thus called *accelerated tests*. The component's degradation process can be accelerated by increasing the temperature, and also by increasing the value of the power supply.

Most Integrated Circuit failure mechanisms are based on physico-chemical reactions that are accelerated by temperature in accordance with the Arrhenius equation. According to the MIL-HDBK-217:

$$\lambda_b = A \exp(-E / (k T)), \text{ where:}$$

- λ_b is the process failure rate (base or intrinsic failure rate depending on the technology),
- E is the activation energy for the process (in electron volts - eV),
- k is the Boltzmann's constant ($8.6 \times 10^{-5} \text{ V} / ^\circ\text{K}$),
- T is the temperature in Kelvin degrees,
- A is a constant,
- \exp is the exponentiation operator.

The real value of the λ of a given circuit is deduced from λ_b by a relation $\lambda = \lambda_b \lambda_1 \dots$ which integrates numerous λ_i factors characterizing the influence of the manufacturing process.

7.2.4 Reliability Evolution

In reality, experiments conducted on physical products (mechanical devices or electronic components) show that the failure rate λ is not constant during time. It is assumed that this number is high at the beginning of the product's life (infant mortality), then it drops and becomes more or less constant during its useful life, and finally it increases substantially during its wearout phase.

This evolution is typically presented by the $\lambda(t)$ curve, which is described as a *bathtub curve*, shown in *Figure 7.4*. So, the hypothesis that λ is constant generally corresponds to the 'active life' stage only.

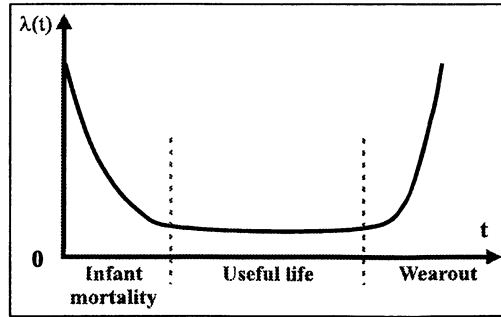


Figure 7.4. Bathtub curve

7.3 TESTABILITY

We should note that the *test* is one of the fundamental means of fault suppression. Used during the design, manufacturing and operation stages, a test consists in subjecting a product or its model (a model during the design, as the product is not yet a physical product) to an experiment from the outside, e.g. by a tester. The test is conducted through functioning sequences which are made of input/output vectors. A test allows:

- to show the presence of faults (*detection test*),
- and eventually to localize them (*diagnosis or localization test*).

We will not develop these test methods here, as they will be analyzed in Chapters 12 and 13.

Testability measures several factors:

- the *ease* with which a given product can be tested, that is to say the facility with which we can determine detection or localization test sequences, and the facility with which these sequences can be applied.
- the *length* of the obtained test sequences, that is to say the number of input vectors to be applied to the product and the number of corresponding output vectors to be observed,
- the *coverage* or efficiency of the obtained test, that is to say the percentage of detected faults in relation to the total number of faults which could affect the product according to a predefined fault model.

A product with a ‘good testability’ allows to rapidly determine a test sequence having a short number of vectors and a high fault coverage. As certain technological choices have an influence on the final product’s testability, design or production choices can increase or decrease the

testability. For example, the injection of observation means in the heart of the product greatly facilitates the detection of errors during functioning (on-line detection), but also during test operations. This is the case of systems which integrate error detecting codes or programs instrumented by executable assertions.

Several methods have been developed in order to evaluate the testability of a product. They are often based on an analysis of the product's structure and an estimation of its *controllability* and *observability*. These two 'system level' notions of *controllability* and *observability* are linked to the ease with which the internal states of the system can be controlled from the input variables, and their real values observed at the external output variables.

Testability is assessed on a product, but also on the used technology. This notion has recently been applied to software technology. For example, the ISO 15942 standard evaluates each feature of the Ada language in terms of the ease of verification of programs using this feature. Each feature thus receives one of 3 grades:

- *included* when the feature use makes the verification easier,
- *allowed* when its using requires additional but tractable work,
- *excluded* when the verification techniques cannot be applied when the feature is used in the program.

Testability and reliability are two different attributes, which are however correlated. Indeed, a good testability has to lead to an increase in the number of faults detected, which therefore leads to a product's higher reliability.

7.4 MAINTAINABILITY

7.4.1 Maintenance

Maintenance is an important activity related to product operation. This activity will be explained in this section in order to introduce the *maintainability* criterion.

7.4.1.1 Definitions

Maintenance is originally an operation linked to the operation stage of a life cycle. It consists in stopping the product's mission and subjecting it, 'off-line', to a certain number of ***troubleshooting and repair*** actions: determination of its state of health (presence of faults by *detection*) and eventually putting it back in a good state by *localization*, and then *repair* of the fault (*Figure 7.5*).

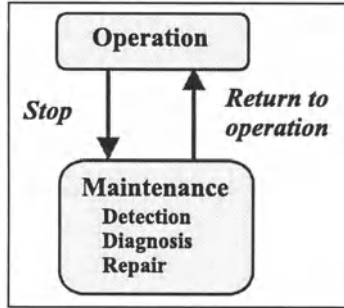


Figure 7.5. Maintenance

The product is therefore said to be *repairable* in the context of its application. However, in numerous cases, maintenance is not applicable because the systems are isolated or inaccessible. This is the case, for example, with the hardware elements of satellites or polar beacons. Such systems are therefore said to be *non-repairable*. Some other products are both repairable and non-repairable according to the different stages of their life. Thus, a rocket is considered to be *repairable* when it is on ground whereas it becomes non-repairable when it is launched.

Maintenance is the set of actions which permit a product to be maintained or re-established in a specified state, or to be ready to deliver a determined service.

7.4.1.2 Maintenance Categories

Three large categories of maintenance exist (Figure 7.6):

- *preventive maintenance* when an action detects the presence of faults before they lead to a failure,
- *corrective maintenance* when a product reputed to be failing is cured,
- *evolutive maintenance* in order to improve the product’s functionality.

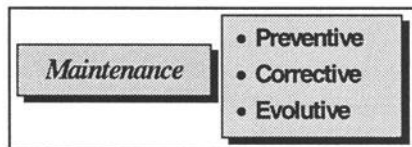


Figure 7.6. Maintenance categories

These three categories call for different techniques and have to resolve different problems, at the technical level as well as at the management level.

Preventive maintenance

Preventive maintenance of computing equipment is carried out with a *fixed* or *variable* periodicity. The measure of the maintenance periodicity does not always use the same units. They depend on the considered application domain. We can use the absolute time, the number of functioning hours or the number of miles covered. Thus, an automobile is revised every 5000 miles or each year.

Two variants of preventive maintenance exist:

- *systematic* (or *scheduled*) preventive maintenance when its occurrence is fixed,
- *conditional* preventive maintenance when it is conditioned by some operational events (use of wearout or temperature sensors, etc.).

Corrective maintenance

Corrective or *curative maintenance* is carried out after the detection of an anomaly during the product's functioning on the mission site. Thus, we drive a car to the garage if it does not function properly.

Evolutionary maintenance

Evolutionary maintenance deals with the modification of certain functions of an already designed and developed product which is supposed to operate correctly. This is done to improve its performance, or to adapt it to new procedures or constraints. For example, the successive versions of a product software: 1.0, 2.0, etc.

7.4.2 Maintainability

7.4.2.1 Definition

Maintainability measures the aptitude of a product:

- to be repaired, that is to say putting the product back into a correct functioning state, suppressing the present fault,
- to evolve, that is to say to accept modification by adding new functionality or improving already existing functionality.

First of all, maintainability aims at measuring the ease with which a preventive or corrective maintenance operation can be carried out: detection and localization of the fault(s), repair and, eventually, re-initialization. Secondly, maintainability assesses the ease with which the product allows the modifications to be done in order to adapt it to a new functional and/or non-functional environment or to accept a new functionality.

Maintainability is an important criterion to assess dependability. In many cases, the increase of its value reduces the risk of fault introduction during the maintenance stage, and hence increases the reliability of the product.

Note. The term *serviceability* is used by numerous manufacturers of electronic components and computing systems in order to express the maintainability.

7.4.2.2 Probabilistic Models

Where the aptitude to repair is concerned, we often define probabilistic maintenance models. In a similar way to the exponential reliability model, the most frequent model is the exponential law with a constant coefficient:

$$M(t) = 1 - e^{-\mu t},$$

M is the probability of being repaired, μ is called the *repair rate*.

From this definition, the *Mean Time To Repair* or *MTTR* is deduced. The MTTR is the average time between the instant of failure occurrence and the return to full functional operation.

$$\text{MTTR} = 1/\mu$$

Where the measure of the product's capacity to evolve is concerned, we quantify its *complexity* by evaluating:

- the degree of *structuration*, for example, in a program, we count the average number of statements in its sub-programs,
- the degree of *coupling*, for example, in a program, we determine the number of shared variables, the complexity of the graph expressing the sub-program calls, etc.

7.4.3 Reliability and Maintainability

Corrective maintenance operations are supposed to integrally restore the functionality of a product. In reality, three cases are often met:

- *stable reliability*: the capability of the product to deliver its service is statistically preserved (the *failure rate* remains constant after repair);
- *increasing reliability*: the failing components are replaced by higher reliability components when design or production faults are eliminated (and without introducing new faults!), thus the failure rate decreases;
- *decreasing reliability*: in some cases, maintenance operations set out to weaken some components which become less reliable, or in other cases, the reliability of components decrease naturally with time, and therefore the failure rate increases (classical wearout phenomenon).

7.5 AVAILABILITY

The availability criterion concerns repairable products, that is to say products which are submitted to destructive and repairing mechanisms.

Availability is the probability that the product functions correctly at time t , knowing that it functions correctly at the initial time.

This attribute differs from the reliability because it takes into account the error correction mechanisms introduced during the development of the product. In the case of simple exponential laws modeling the degradation (faults) and repair mechanisms, we use probabilistic finite state machine models whose arcs have been labeled by λ and μ coefficients. *Figure 7.7* shows a simple state diagram whose state 1 expresses that the product operates correctly (hence, the product is *available*), state 2 specifies a failing situation. The arc 1-2 models a failure of the product with a failure rate λ (probability labeling the arc); the arc 2-1 represents a repairing of the failing product with a repair rate μ (probability labeling the arc).

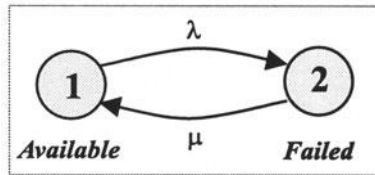


Figure 7.7. Degradation / repair cycle

Instant availability. For exponential failure and repair laws with constant failure rate λ and repair rate μ , the *instant availability* is:

$$A(t) = \mu / (\mu + \lambda) + \lambda / (\mu + \lambda) \cdot e^{-(\mu + \lambda) t}$$

We should note that the availability is strictly equal to the reliability of a non-repairable system ($\mu = 0$). In the opposite case, it is superior. Indeed, the probability of functioning correctly at time t is increased by the repair mechanism: a fault arriving between 0 and t could have been treated and the product therefore returned to its correct state at t .

Availability during the permanent stage. The *permanent availability* is defined in the permanent stage (when such stage exists):

When $t \rightarrow \infty$, $A(t) \rightarrow A = \mu / (\mu + \lambda)$, i.e. $A = MTBF / (MTBF + MTTR)$.

We also define two other estimators in permanent functioning:

- the **Mean Down Time (MDT)**, mean time during which the product is not available,

- the *Mean Up Time (MUT)*, mean time during which the product is useable.

Example 7.1. Repairable Product

In order to illustrate the relationships between the notions of reliability, testability and availability, we consider a repairable product. At time t , this product is affected by a fault which transforms itself into an error, then a failure, before being detected, for example, by application of a test sequence. Then, we use a diagnosis technique which localizes the fault, then repairs it and puts the product back in service.

Figure 7.8 shows this mechanism: we see the periods of availability (*Mean Up Time*) and non-availability (*Mean Down Time*). The period of the correct operation depends on the product reliability. The duration used to detect and localize the existing faults depends on the product maintainability, more precisely its testability. The time spent in correcting the faults depends on the product maintainability. All these characteristics affect the product availability. Exercise 7.1 refines the study of this diagram.

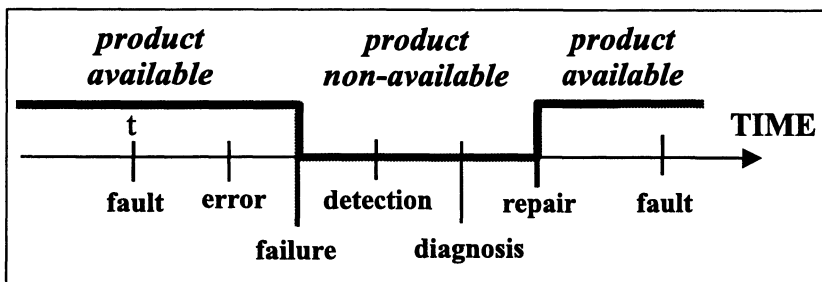


Figure 7.8. Example of a repair cycle

7.6 SAFETY

Safety is directly linked to the notion of seriousness of the failures described in the first part. We have explained that the failures induce several classes of external consequences (Chapter 4): benign, significant, serious, catastrophic. Safety is the privileged criterion for highly critical applications for which the consequences of certain failures are catastrophic: embedded systems from avionics or space domain, etc. This criterion measures the trust which can be attributed to a product which does not present failures having catastrophic external consequences.

Safety is the probability that the product will not have failures belonging to unacceptable seriousness classes, between the initial time and a given time t .

Very often, the unacceptable seriousness classes concerns *catastrophic failures*. If we refer to a statistical state model, we measure the probability of not reaching the third state, which is judged to be unacceptable, knowing that it was in state 1 at the initial time and that we know the probability of passing from state 1 to states 2 or 3 (*Figure 7.9*).

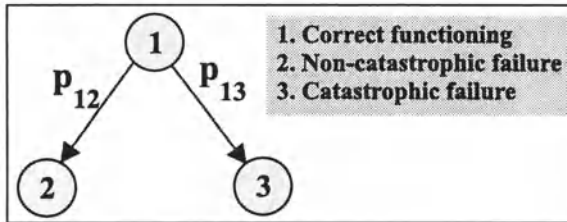


Figure 7.9. Safety: dangerous failing state

A simple example is that of a balloon's heating regulator which contains a dangerous liquid. A failure of this regulator is catastrophic if it leads to the explosion of the balloon by overheating. We measure the safety of this product as the probability that it will not reach a state that provokes an explosion. The safety naturally depends on the technology used and the environment's parameters (such as temperature), but also on the protection mechanisms which tend to avoid the occurrence of the failure which causes an explosion (by a suitable design process avoiding the fault presence and/or occurrence), or to prevent the failure from provoking an explosion (by an external product protection or, more generally, a fault tolerance mechanism).

Relation between safety and reliability

A system that continues to function correctly for long period of time has a good reliability. However, it is possible to have reliable but unsafe systems as well as safe but unreliable systems. A safe system can fail, as long as it does so without creating an accident: destruction of the controlled process or human injuries or deaths.

A handgun may be very reliable but particularly unsafe. In many systems, safety and reliability go hand-in-hand. For example, reliability is a very necessary safety condition for an aircraft, as most of the failures of the flight control system may have catastrophic consequences.

Methods that increase safety are of course expensive! A first obvious approach concerns the increase of reliability of the components used: this

produces fault probability reduction, therefore failure reduction. However, when this approach turns out to be insufficient, we use specific redundancy techniques. They can at the origin of some antagonistic effects between the two reliability and safety parameters: for instance, the increase of the number of components destined to increase the safety often reduces the reliability. In sub-section 7.9.3 we will see how to apprehend the quantitative analysis of safety on *Markovian* type models.

7.7 SECURITY

Security is an attribute of dependability with regard to the prevention of unauthorized access and/or handling information.

This attribute covers two parameters: *confidentiality* and *integrity*:

- **Confidentiality** measures the non-occurrence of unauthorized disclosure of information.
- **Integrity** expresses the non-occurrence of improper alterations of information.

These parameters lead to numerous techniques to protect the product's access, or its utilization. The simplest example of confidentiality means is the use of passwords in order to access a computer. We also use encryption techniques to protect data from being understood in case of involuntary or fraudulent accesses. The security attribute is not considered in this book.

7.8 SYNTHESIS OF THE MAIN CRITERIA

The evolution of a product's functioning is symbolized in *Figure 7.10* by a simple probabilistic three state model.

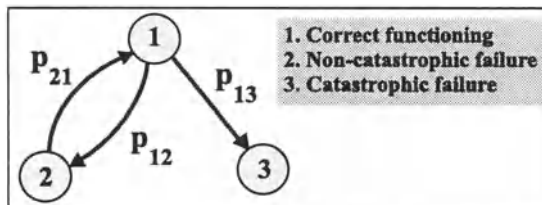


Figure 7.10. Probabilistic model

State 1 is the correct functioning state, state 2 is a failing state which does not lead to the loss of the mission and can be repaired according to the arc

(2-1), and state 3 is a catastrophic failure state. The arc (1-2) that leads to the non-catastrophic failure of the product is labeled by the failure rate p_{12} , the arc (2-1), which restores the correct function, is labeled by the repair rate p_{21} , and the arc (1-3) which lead to the end of the mission is labeled by the rate p_{13} . We should note that the probabilities associated with the transitions are expressed in general with failure and repair rates of type λ and μ .

With this basic model, the four principal dependability criteria are expressed by *Figure 7.11*, the graph being in state 1 at the beginning. Expression $q(t)$ represents the state in which the system is at time t . At the initial time ($t = 0$), the system is supposed to be in the correct state 1.

RELIABILITY:	$R(t) = P(q(\tau) = 1, \tau \in (0, t))$
SAFETY:	$S(t) = P(q(t) \neq 3)$
AVAILABILITY:	$A(t) = P(q(t) = 1)$
	Repairable Systems
MAINTAINABILITY:	$M(t) = P(q(t + \Delta) = 1 q(t) = 2)$
	Repairable Systems

Figure 7.11. Expression of the main attributes

- The **reliability** at time t is the probability (noted as P in *Figure 7.11*) that the product remains in state 1 from time 0 to t . This corresponds to a measure of the capacity that the product does not fail between 0 and t .
- The **safety** at time t is the probability that the product is not in state 3 at time t . This property implies that the product will never reach state 3 between 0 and t .
- The **availability** is the probability that the product is in state 1 at time t , whatever the evolutions which occurred before time t . The product may produce non-catastrophic failures which are repaired: its state changes between states 1 and 2.
- The **maintainability** is the probability that the product failing at time t will be repaired before a certain predefined Δ duration. This definition is a variant of the exponential law with a constant failure rate; it emphasizes the repair delay.

Let us note that the two last criteria apply to repairable products.

Table 7.1 synthesizes the four attributes, reliability, availability and maintainability, for exponential laws with constant coefficients applied to a

repairable system. Note that availability expression is equal to reliability expression when $\mu = 0$.

These attributes must be precisely defined in the specifications of a project leading to any industrial product. According to the application, one or several attributes can have a particular importance, leading to the use of appropriate development techniques. Reliability is essential for a spatial probe (for example, $R = 0.99999$ after a 12 month mission); as no repair is possible, maintainability has no sense and availability and reliability laws are equal. The development of a telephone electronic switching system requires high availability (for example, a few minutes of unavailability per year). The first attribute of a control system embedded in an aircraft is safety (for example, 10^{-9} catastrophic failure during a flight). Naturally, in the general case, a compromise must be found between the dependability requirements expressed by quantitative values of the previous attributes, and the other criteria of the specifications (cost, development duration, etc.).

Reliability	$R(t) = e^{-\lambda t}$ $MTBF / MTFF = 1/\lambda$.
Availability	$A(t) = \mu / (\mu + \lambda) + \lambda / (\mu + \lambda) \cdot e^{-(\mu + \lambda)t}$ $A(\infty) = A = \mu / (\mu + \lambda) = MTBF / (MTBF + MTTR)$
Maintainability	$M(t) = 1 - e^{-\mu t}$ $MTTR = 1/\mu$

Table 7.1. Simplified expressions of the main attributes

7.9 QUANTITATIVE ANALYSIS TOOLS AT SYSTEM LEVEL

In the following paragraphs we introduce three models and methods used for quantitative analysis: the *fault simulation*, the *reliability block diagrams* which constitute one of the first analytical models used (see also in Appendix B), and the analysis of *non-deterministic state graph models* (such as *Markovian graphs*).

7.9.1 Fault Simulation

Fault simulation constitutes a universal approach, intensively used in different situations. It assumes an 'executable' system model of the product studied, a set of external input/output sequences which are applied to this model, and the possibility to inject faults of a fault model in the system model. This is why some of these techniques are called *fault injection*.

We will briefly explain the principles of the *Monte Carlo simulation*, which is a relatively simple and easy process. The events which make the system evolve are the destructive and repairing mechanisms. At each step of simulation, these events are randomly chosen and injected whilst taking their respective probability laws into consideration.

This process is repeated a certain number of times, starting from the same initial state. The statistical laws will make the system evolve towards different states which are recorded. If the number of simulations carried out is sufficiently important to satisfy the law of large numbers, we can deduce from this simulation significant quantitative information about the dependability parameters such as reliability and availability. For example, we calculate the number of favorable cases among the total number of cases, in order to estimate the survival or safety of a simulated product. This method requires a system behavioral model and often implies long run-time. However, it is a very flexible method, accepting complex statistical models and the introduction of queuing mechanisms used in computing to access certain resources.

7.9.2 Reliability Block Diagrams

Once a product has been designed from an assembling of elementary components with known reliability, the global reliability of the product can be deduced. The *reliability block diagram* method introduced hereafter comes from studies on electronic components. However, it is also used for reliability studies done at system level.

7.9.2.1 Series Reliability Model

Consider a product constituted by n components, C_1, \dots, C_n , having reliability laws, $R_1(t), \dots, R_n(t)$. Let us assume that the failure of one of them is sufficient to provoke a product's failure (this is the case with the majority of products). The reliability of the whole system is then derived from the reliability of each component, by using the classical theorem of independent probabilities. Hence, the global reliability is the product of the reliabilities of the components: $R = \prod R_i$.

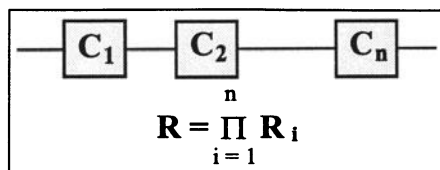


Figure 7.12. Components 'in series'

When the reliability of the components is defined by exponential laws with constant failure rates, the reliability of the global product is also defined by an exponential law with a failure rate λ which is the sum of the failure rates of the components (λ_i): $\lambda = \sum \lambda_i$.

Therefore, it is said that these components are 'in series' and we establish a *reliability block diagram* shown in *Figure 7.12*. Exercise 7.3 develops these calculations and establishes that, when the components are all identical (each having a rate λ_0), the global failure rate is multiplied by n , and the global *MTBF* is divided by n :

$$\lambda = n \lambda_0, \text{ and } MTBF = MTBF_0 / n, \text{ with } MTBF_0 = 1 / \lambda_0.$$

Consequently, the reliability decreases according to the number of components: it is sensitive to the *complexity* with regard to this number.

7.9.2.2 Parallel Reliability Model

All electronic structures are not of the previous 'series' type. In certain redundancy cases, the failure of the product only appears when all the components are failing. A simple example is that of two light bulbs in parallel: as long as one light bulb functions correctly, lighting is ensured. We often meet such redundant structures in dependable products, because they have a better reliability. They also have a better availability in the case of repairable systems.

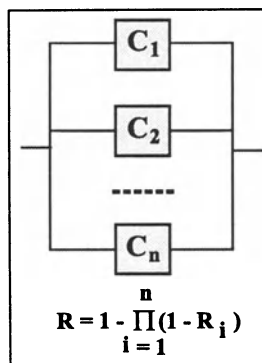


Figure 7.13. Components in 'parallel'

For n components, the reliability block diagram of this situation is represented in *Figure 7.13*: we say that these components are in 'parallel'. The reliability increase is due to the fact that the probability that the product fails is the product of the failure probabilities of all components;

$$1 - R = \prod (1 - R_i) \rightarrow R = 1 - \prod (1 - R_i)$$

Exercise 7.3 provides the opportunity to carry out calculations and to

show that the reliability is increased: when the failure rates of the components are identical, and for $n = 2$, we show that the MTBF is multiplied by 1.5 only. Thus, the improvement of reliability is not proportional to redundancy means: it is always smaller.

We will meet more sophisticated redundancy situations, such as a redundancy with different reliability values of duplicate modules, or redundancy with majority vote. The study of the reliability of these structures involves other tools not considered here, such as the *Laplace* transform.

In Exercise 7.4, we will compare two reliability block diagrams: a 'parallel-series' structure versus a 'series-parallel' structure.

To complement this introductory section, some results of the reliability of classical structures are given in Appendix B.

7.9.3 Non-Deterministic State Graph Models

A *Markov graph* is a state graph with non-deterministic transitions. The behavior evolves from state to state as in a classical state graph, but the transitions between states are labeled by probabilistic values. The fact of firing one transition or another one from a given state is in relation to these probabilities. In dependability studies, such graphs express the different states of a product submitted to degradation and protective mechanisms. As long as some mathematical hypotheses are satisfied, we can apply simple analysis tools handling probabilistic matrix associated with these graphs.

Therefore, we evaluate a product's behavior in terms of the probability to reach or not a state or a group of states from an initial known state (typically the fault free state). A matrix analysis method based on a *Markov* graph in introduced by Example 7.2.

Other methods based on non-deterministic state graph models have been defined, such as the analysis of *stochastic Petri nets*. This model has the property to express parallelism. It uses *places* and *transitions*; *tokens* are placed in some places and this marking constitutes the *state* of the Petri net at a given time. Firing rules allows an asynchronous evolution of the graph. Example 7.3 shows the use of this model to represent the evolutions of a redundant system.

Example 7.2. Analysis of a simple graph

We consider a repairable product with constant failure and repair rates. *Figure 7.14* shows its evolution graph which specifies the states and the transitions between these states. The evolution modeled by this graph is discrete with time; the time unit is the unity of failure or repair rate time: for example one hour. This evolution is controlled by the probabilities which

label the arcs. We remark that the sum of the probabilities of all the arcs leaving the same state is always equal to 1.

Suppose that the product is in state 1 at time n :

- the probability that it stays in state 1 at time $(n + 1)$ is $(1 - \lambda)$,
- the probability that it goes to state 2 at time $(n + 1)$ is λ .

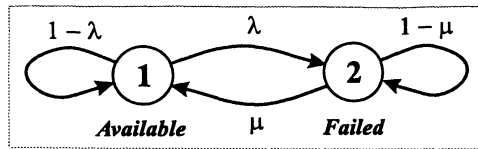


Figure 7.14. Simple Markov graph

We associate a two-dimensional evolution matrix with this graph:

$$P = \begin{bmatrix} 1 \rightarrow 1 & 1 \rightarrow 2 \\ 2 \rightarrow 1 & 2 \rightarrow 2 \end{bmatrix} = \begin{bmatrix} 1 - \lambda & \lambda \\ \mu & 1 - \mu \end{bmatrix}$$

Example of numerical values: $\begin{bmatrix} 0.9 & 0.1 \\ 0.8 & 0.2 \end{bmatrix}$

Each element indexed by ij gives the probability of passing in state j from state i .

If P is squared, the resulting matrix gives the probability to reach, in the next elementary time, state 1 or state 2 from an initial state 1 or 2. By calculating the successive raising of P to the power of n , we analyze the evolution of these probabilities when time progresses:

$$P = \begin{bmatrix} 0.9 & 0.1 \\ 0.8 & 0.2 \end{bmatrix}, \quad P^2 = \begin{bmatrix} 0.89 & 0.11 \\ 0.88 & 0.12 \end{bmatrix}, \quad P^3 = \begin{bmatrix} 0.889 & 0.111 \\ 0.888 & 0.112 \end{bmatrix}$$

As the system is initially supposed to be in the correct state 1, we want to know the evolution probabilities towards the correct state 1 or the incorrect state 2, with time. The probability of being in the incorrect state is 0,11, then 0,111. A permanent state is reached when these probabilities stabilize.

We can also study Markovian processes with continuous evolution, the transitions between states being continuous probabilities during time.

Example 7.3. Stochastic Petri net

A regulation system has three redundant active units and one inactive spare unit. The regulation function is ensured as long as two of the active units have no failure. When one unit is faulty, a reconfiguration process is started: this process replaces the faulty unit by the spare unit, if this unit has

not already been used.

This system can be represented with the Petri net of *Figure 7.15*. It has three places: *P1* represents the active units by the number of tokens which are inside, *P2* represents the spare unit, and *P3* represents the failing units. The graph is initialized with 3 tokens in *P1* (three active units), 1 token in *P3* (one spare unit), and 0 token in *P2* (no failing units).

Transition *T1* is labeled with the failure rate λ of the active units. When one unit fails, one token is taken in *P1* and one token is set in *P2*. In this state, the system continues to function correctly. If a second active unit fails, before any restoration, a second token is shifted from place *P1* to place *P2*, and the system fails. Transition *T2* is labeled with the restoring rate ρ . If one spare unit is available (one token in *P3*), and if there is at least one token in *P2*, then *T2* is fired: one token is removed from *P2* and *P3*, and one token is put in each place *P1* and *P2*. Hence, the regulation system works again. Now, if another active unit fails, the system will fail and the restoring mechanism will not be possible, as no token remains in *P3*.

Exercise 7.6 proposes to modify this Petri net according to a changing in the specifications.

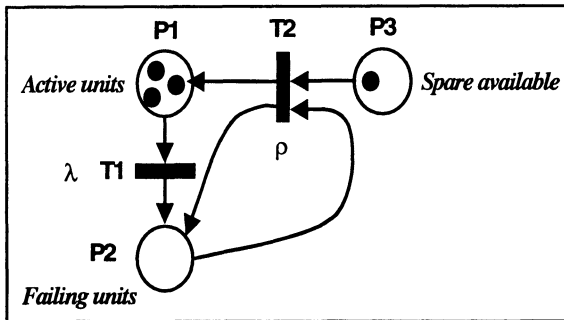


Figure 7.15. Stochastic Petri net

7.10 INDUCTIVE QUALITATIVE ASSESSMENT: FAILURE MODE AND EFFECT ANALYSIS

7.10.1 Principles

The *FMEA (Failure Modes and Effects Analysis)* is a normalized technique dedicated to *qualitative analysis* of reliability and safety. It is based on an *inductive* process, which starts with simple failures (altering components or modules) in order to deduce their consequences on the complete system. This approach is used in numerous fields such as avionics,

aeronautics, nuclear, chemical and automotive industries.

Initially developed by the US army in 1949 (military procedure MIL-STD-1629A, 'Procedure for Performing a Failure Mode, Effects and Criticality Analysis), this approach has then be amended and refined by several institutions: CEI document 812-1985, AIAG (Automotive Industry Action Group) and ASQC (American Society for Quality Control) in 1993, SEA (Society of Automotive Engineers) procedure J-1739.

The AIAG presents this technique as a systematic group of activities intended to:

- recognize and evaluate the potential failure of a product or process and its effects,
- identify actions which could eliminate or reduce the chance of the potential failure occurring,
- document the process.

The keywords on which the FMEA and its main extension, the FMECA introduced hereafter, are based are: the functions, the failure modes, the effects and their severity, the causes and their occurrence and the controls. Hence, FMEA is a technique used to highlight the consequences of a failed component of the system on the behavior of the whole system.

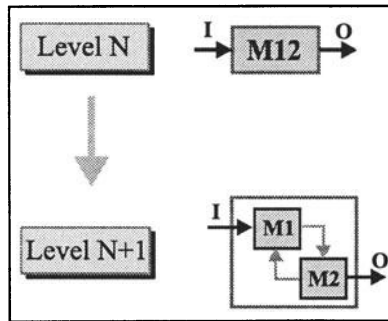


Figure 7.16. Top-Down design

Let us consider a system designed according to a top-down approach. At step N , the designer analyzes the specifications of the components used in the system modeling of this level and proposes their implementation combining sub-components of level $N+1$. Figure 7.16 illustrates this step of the design of one component. Then, the designer defines the possible failures of the sub-components.

The FMEA deals with the study of the effects of known failures at level $N+1$, then N , ..., and finally at the global system level and the environmental level. The error propagation analysis is expressed by tables providing:

- the identity of the analyzed component (name, reference, number, etc.),
- the function performed by the component,
- the considered failure of the component, i.e. an error of the global system,
- the possible cause of the failure (this is optional),
- the local effects, that is to say the consequences on the others components at the same design level ($N+1$),
- the effects at the next higher level (N), i.e. the component which includes the failed component,
- the effects on the global behavior (end effects),
- the failure detection method,
- the reaction to the errors.

These last two aspects are relative to the techniques used to handle the errors. They will be considered in Chapter 17 (fail-safe systems) and Chapter 18 (fault-tolerant systems).

7.10.2 Means

The MIL-STD-1629 standard defines a worksheet used to express all the necessary pieces of information (*Figure 7.17*). The filling of this worksheet reveals two problems: the definition of the failure modes, and then, their propagation to highlight their effects on the components introduced at each design step, and finally on the product's services.

System _____					Date _____						
Indenture level _____					Sheet _____ of _____						
Reference drawing _____					Compiled by _____						
Mission _____					Approved by _____						
Ident. number	Item/functional identification (nomenclature)	Function	Failure modes and causes	Mission phase/operational mode	Failure effects			Failure detection method	Compensating provisions	Severity class	Remarks
					Local effects	Next higher level	End effects				

Figure 7.17. FMEA Worksheet

At low level of hardware system design, quite simple and realistic error models exist, which can be used as pertinent failure modes. During the first

steps of a system design, the failure modes are defined as violation of properties on the use (pre-conditions) or the behavior (post-conditions) of the components. The considered errors must be representative of actual errors, that is to say errors that can occur, taking the component design into account. This can be concluded by using a deductive analysis (c.f. Fault Tree Method in next section). Moreover, nobody knows if the list of the studied errors is complete. To cover all the errors, general properties must be considered. For instance, “the actuator provides a bad value” is better than “the actuator provides value $V1$ instead of V ”, as the numerous other values, $V2$, $V3$, etc., will not be handled.

The second problem deals with the propagation of errors through the system structure. When failure modes are defined by bad values, and the system modeling tool is formally defined, a simulation provides the effects. Such a situation occurs, for instance, for a stuck-at 1 error of a component of a structure defined by interconnected gates. On the contrary, when failure modes are defined by temporal properties (such as “the data output is delayed”) or by general properties (such as “the data output is incorrect”), the assessment of their effects is more difficult. This difficulty also exists when the modeling tool used to express the system does not possess formal semantics. This occurs when the relationships between the components are expressed in English.

7.10.3 FMECA

The *failure modes* are potential failures of components. When their occurrence probability is known, it is possible to deduce the probability of occurrence of failures at the global system level. **FMECA** (**Failure Modes, Effects and Criticality Analysis**) is a variant of FMEA that associates a probability with the failure of the components and with their effects. Hence a seriousness class and its occurrence risk can be associated with each failure.

As previously mentioned, the values of the probability of the initial errors (failure modes) are generally obtained by exploitation feedback. Some of these values are standard for a given technology: for example a semiconductor manufacturer provides the user with the failure probability of an integrated circuit. Other values are specific to each design process: it is the case of design faults which are influenced by several parameters such as the tools, the design ‘style’, the used methods, the design team, etc. Thus, FMECA is a qualitative as well as a quantitative method.

7.11 DEDUCTIVE QUALITATIVE ASSESSMENT: FAULT TREE METHOD

7.11.1 Principles

Numerous failures can be imagined. Fortunately, only a few of them may occur. The *FTM (Fault Tree Method)* aims at examining if a supposed failure may occur or not, taking the system structure into account. It also defines the circumstances of the failure occurrence, by expressing the studied failure as a composition of run-time events using the ‘AND’ and ‘OR’ operators as shown in *Figure 7.18*

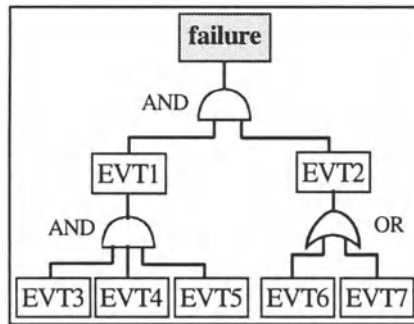


Figure 7.18. Fault Tree Method

This figure specifies that the failure is raised if (*EVT1* and *EVT2*) occurs. Then, it explains the causes of the occurrence of *EVT1* (*EVT3* and *EVT4* and *EVT5*) and *EVT2* (*EVT6* or *EVT7*). Three situations allow to conclude on the failure effectiveness.

- If the values of the leaves of the fault tree (*basic events*) are known, the failure raising can be predicted. For instance, if *EVT3* = false, and *EVT4*, *EVT5*, *EVT6* and *EVT7* are true, then the failure cannot appear.
- Relationships between the basic events show contradictions. For instance, suppose we know that *EVT5* is true only if *EVT6* and *EVT7* are both false. Thus, *EVT1* and *EVT2* cannot be true simultaneously, and consequently, the failure cannot occur!
- Relationships between the events of a branch reveal contradictions. For instance, *EVT5* is the negation of the assertion defining the failure.

In practice, the three studies are often combined and the conclusion on the failure occurrence is not simply *true* or *false*, but a potentiality. However, the fault tree defines the circumstances of this potentiality.

Two difficulties exist in the definition of fault trees: the choice of the failures to be examined, and the obtaining of the tree from a given failure. The tree is built by a system structure analysis. When the formal composition laws can be used to combine elements to define a structure, a systematic process is sometimes proposed to derive the tree. For instance, Nancy Leveson proposes such process when Ada programming language is used to express a software system.

Note. The nodes of the *fault tree* being general events, including erroneous but also correct events, this modeling is also called *event tree*.

Example 7.4. Redundant system

A system is made of three modules: *M1*, *M2* and *M3*. *M1* and *M2* are two redundant active units: as long as one of them is faultless, the performed function is correct. *M3* ensure another part of the system's function.

Thus, the system fails if *M1* and *M2* fail or *M3* fails. This analysis can be represented by the fault tree of *Figure 7.19*. This model can be used to perform quantitative failure evaluations. Exercise 7.7 proposes to evaluate the reliability of this system with the FTM approach, and to compare with the Reliability Block Diagram approach.

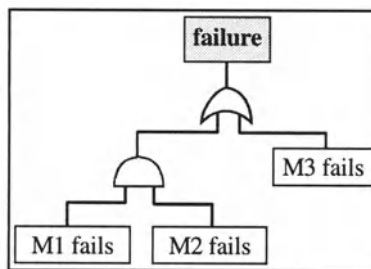


Figure 7.19. Redundant system

7.11.2 Software Example

Example 7.5. Stack

Consider the procedure `Simple_Example` which uses the procedures `Push` and `Pop` provided by the package `Stack`:

```

with Stack, Ada.Text_Io;
procedure Simple_Example (Element: in out
                          Type_Element) is
begin
  Stack.Push (Element);
  Stack.Pop (Element);

```

```

exception
  when Overflow ==> Ada.Text_Io.Put_Line
    ("Stack Overflow");
end Simple_Example;

```

The exception `Overflow` (respectively `Underflow`) can be raised by the procedure `Push` (respectively `Pop`). The exception `Overflow` is handled locally by the procedure `Simple_Example`, whereas no handler exists for `Underflow`. So, if it is raised by `Pop`, this exception is propagated by `Simple_Example` signaling a failure. We use the Fault Tree Method to study this failure.

The exception `Underflow` is raised by `Simple_Example`: if it is raised by `Push`, or if it is raised by `Pop` and no exception is raised by `Push`.

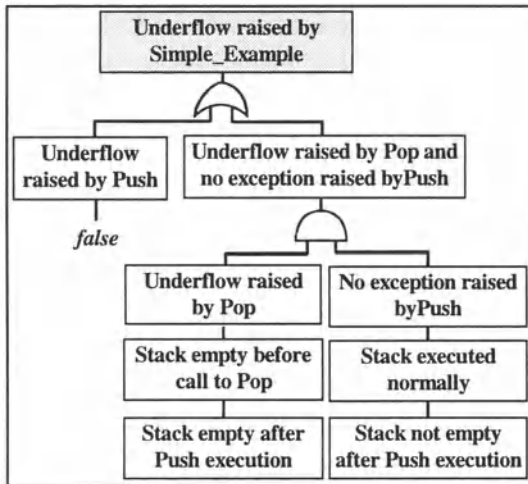


Figure 7.20. Example of Fault Tree

This last condition is due to the fact that an exception raising terminates the execution of the procedure body. The top of *Figure 7.20* illustrates this first step of analysis.

Looking at the body of the procedure, `Push` (not provided here), we notice that it cannot raise the exception `Underflow`. So, the event “Underflow raised by Push” is always false.

The event “Underflow raised by Pop and no exception raised by Push” results from two branches connected by a AND.

- The `Underflow` can be raised by `Pop` if the stack is empty before `Pop` is called. This conclusion is obtained by analyzing the `Pop` procedure body (not provided here). As `Pop` is called after `Push`, this conclusion is derived from the assertion “Stack empty after Push execution”.

- If no exception is raised by `Push`, the procedure execution was concluded normally. So, an element was memorized in the stack which is therefore not empty.

In conclusion, an `Underflow` can be raised by `Simple_Example` if E and not (E) is true, where $E = \text{“Stack empty after Push execution”}$. This contradiction leads to conclude that the root event is false; hence the considered failure cannot occur.

7.11.3 Use of the FTM

Generally, the Fault Tree Method does not lead to conclude that a failure is always true (that is, the product is always failing), or always false (the failure will never occur). This method provides a Boolean expression which defines the cause of the failure. For instance, *Figure 7.18* specifies that:

$$\text{failure} = (EVT3 \cdot EVT4 \cdot EVT5) \cdot (EVT6 + EVT7),$$

where ‘.’ and ‘+’ represent the AND and OR operators.

Partial knowledge on the basic event values allows to reduce the expression. For instance, assume that $EVT4$ is always true and that $EVT6 \Rightarrow EVT7$. Then, this expression becomes:

$$\text{failure} = EVT3 \cdot EVT5 \cdot EVT6$$

This result will be used by most of the fault handling techniques. For example, the occurrence of the basic events must be prevented, or the faults that make the expression true must be detected and removed. This expression also gives the circumstances of the failure, information very useful to design a fault-tolerant product.

Let us note again that this method has also quantitative assessment applications. Indeed, if probability values are associated with the basic events, it is possible to apply probability compositional rules (to treat the OR and AND nodes) in order to deduce the probability of any event in the tree, including the failure occurrence probability. *Exercise 7.7* uses this method to calculate the reliability of a redundant system, and to compare the results with those obtained by the *reliability block diagram* method.

7.12 EXERCISES

Exercise 7.1. The ‘fault – error – failure – detection – repair’ cycle

In *Figure 7.8*, place time intervals which correspond to fault latency, detection mean time, then repair mean time. By supposing that the temporal diagram results from a statistical study of the product behavior during

several cycles of functioning, how can the MTBF (for repairable system) and the MTTR be deduced?

Exercise 7.2. Reliability of a component

An electronic circuit has an exponential reliability with a constant λ rate.

1. Calculate the mean time (MTBF or MTTR), as well as the $R(t)$ value at this mean time. Numerical value: $\lambda = 10^{-6}$.
2. Demonstrate that this MTBF/MTTF corresponds to the time which is at the intersection of the tangent at the origin of the curve with the time axis (as indicated by Figure 7.3).
3. Explain why λ is similar to a 'failure rate by unit of time'.
4. With another technology, the component has a failure rate equals to 10^{-7} . What is the relationship of the probabilities of these two versions when $t = 10^4$ hours?

Exercise 7.3. Composed reliability

We wish to study the reliability of a system constituted of 2 basic modules (noted M_i) interconnected according to diverse 'series' and 'parallel' reliability diagrams. The reliability of each module is modeled by an exponential law with a constant failure rate λ : $R_i(t) = e^{-\lambda it}$.

1. Determine the reliability of a 'series' reliability diagram of these modules. Calculate the global MTTF. Study the particular case where the two failure rates are identical.
2. Carry out the same study for a 'parallel' diagram.
3. Consider the previous questions with $\lambda = 10^{-4}$, and compare the reliability of these structures at time $t = 1000\text{H}$.

Exercise 7.4. Comparison of two redundant structures

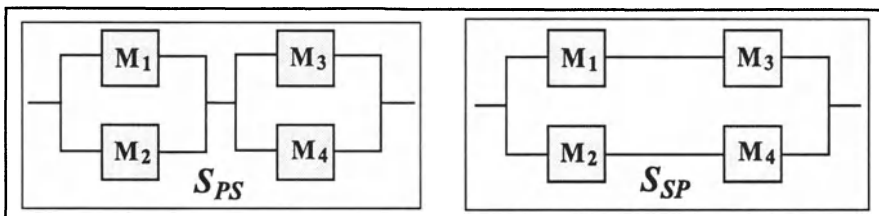


Figure 7.21. 'Parallel-series' and 'series-parallel' structures

1. Study the reliability of the two systems which are represented by the two

reliability diagrams in Figure 7.21, noted as S_{PS} and S_{SP} , knowing that all the modules have the same reliability.

2. Which of the two organizations has the best reliability?

Exercise 7.5. Safety analysis on a Markov graph

Consider the graph in Figure 7.22. From the initial state 1 which represents a behavior without fault, the system degrades with the appearance of faults; it evolves towards the states 2, 3 and 4 which are failing states. On the contrary, protective and repair mechanisms are going to make the system evolve towards better states, for example the state 3 towards the state 1! The arcs between states indicate the hourly rate of evolution (probabilities): $p1$, $p2$, $p3$, $p4$ for the degradations and $r1$, $r2$ for the corrections and repairs.

Study the evolution of the graph from state 1 towards state 4 which is here supposed to be catastrophic.

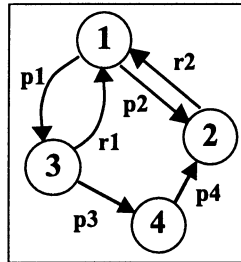


Figure 7.22. Markov graph

Exercise 7.6. Representation of a system by a stochastic Petri net

We modify the specifications of the system studied in Example 7.3. The spare unit has a failure rate λ_s and a repair rate μ_s . When an active unit has been detected faulty, this unit is repaired with a rate μ_a ; this repaired unit can then replace the spare unit.

Represent this redundant system with a stochastic Petri net.

Exercise 7.7. Fault Tree and Reliability Block Diagram

Use the FTM to calculate the reliability of the redundant system of Example 7.4. Compare with the Reliability Block Diagram method.

THIRD PART

FAULT AVOIDANCE MEANS

During the first part of this book we identified the sources of the problems which can affect a product in its applicative environment. In the second part, we firstly introduced the approaches allowing faults and their effects to be mastered: fault prevention, removal and tolerance techniques used or acting during a product life cycle. Then, the dependability assessment means were presented. Finally, we developed the basic notions relative to redundancy, which are necessary to implement the means allowing dependability impairments to be resolved.

In the third and fourth parts we will refine the methods and techniques which allow us to get rid of faults and their internal and external effects. The groups of technique relative to fault prevention and fault removal, known as fault avoidance, are presented in this third part, as they are closely tied to one another. The groups of techniques relative to fault tolerance will be studied in the fourth part of this book.

The writing of these chapters was rather problematic from the author's point of view: how detailed should the presentation be? For example, just the functional and structural testing methods, which provide dynamic analysis of systems in order to detect faults, could by themselves justify an entire book. However, a too detailed description of this subject (as with the others) would not offer an overview of the problems and means of the dependability, which is the aim of this book. We have therefore had to maintain a good equilibrium between the principles and the detailed techniques.

The first two chapters of this part are dedicated to the avoidance of functional faults during the specification (Chapter 9) and during the design (Chapter 10). Then, Chapter 11 deals with the prevention of technological faults. The last three chapters offer a more detailed analysis of the techniques to remove technological faults: an overview of the problems and solutions in Chapter 12, the development of some significant techniques in Chapter 13, and an introduction of design for testability techniques in Chapter 14.

Chapter 8

Redundancy

Whether in the form of traditional error detecting and correcting codes used in transmission systems, in the form of more specific codes such as the *m-out-of-n* code or arithmetic codes, or even in the form of task duplication techniques with majority vote or *type* feature of programming languages, *redundancy* is omnipresent in almost all dependability techniques. In this chapter, we introduce this concept and the associated notions. They will be used later on in the following chapters of parts three and four.

Whether natural, intrinsic, or on the contrary, artificial (e.g. introduced during design), redundancy is a universal property of all systems, independently from their functionality. It can be found in computing, linguistic and biology domains for example. Redundancy basically concerns system's structures by adding more components than necessary. However, it also concerns their behavior, i.e. the input-output relationships, the meaning of human language sentences, or also the semantic of the statements of a programming language. We will observe that the word redundancy is sometimes ambiguous. On the one hand, it can have a pejorative meaning by qualifying that is useless, or even harmful to the dependability. On the other hand, it has sometimes a positive meaning by allowing the detection and/or correction or else the compensation of errors.

We will analyze the two fundamental forms of redundancy:

- *functional redundancy*,
- *structural redundancy*.

We will discuss the possible applications of these forms in order to detect, correct, or tolerate faults. These applications will be described in the third and fourth parts of the book.

8.1 FUNCTIONAL AND STRUCTURAL REDUNDANCY

8.1.1 Linguistic Redundancy

The word *redundancy* comes from the Latin word *redondare* which means plentiful, overflowing. Therefore, the meaning tends towards excess and what is superfluous. Thus, the attribute *redundant* often qualifies what is useless. We are therefore very far from the objectives of the dependability. Its interpretation with a positive sense is actually very recent, principally with the use of error detecting and correcting codes for transmissions.

We say that a product or system presents *redundancy* if some of its constitutive *elements* are not necessary to perform the *normal* input/output relationships.

We will define the two terms *elements* and *normal input/output relationships* in the case of computing systems. But before that, in order to introduce the two forms of the redundancy, we analyze some examples coming from the linguistic domain.

The sentence '*men men are are mortal mortal*' clearly demonstrates a structure redundancy, known as *syntactic redundancy*. This redundancy does not however affect the understanding of the sentence. Naturally, the non-redundant phrase is: '*men are mortal*'.

We consider the following set of three sentences:

Socrates is a man,
men are mortal,
Socrates is mortal.

Any of these sentences read individually does not present syntactic redundancy, however, the third one is semantically implied by the two others: this is a case of *sylogism*. Thus, we notice a second type of redundancy known as *semantic redundancy*.

Many words in human languages can be modified without harming their understanding; to remove for example, a letter *r* in the word *terrible* will not change the meaning at all. This type of redundancy is considered as useful or non-useful, according to the understanding of the person who reads the sentence: this person is known as the *receptor*. Removing these letters only reduces the readability. For example: *men ar mortl* is generally more difficult to understand, but it remains understandable.

All these redundancies induce a growth in the cost of syllables, words, or enunciation time. They have appeared spontaneously in all human languages for varied reasons, logical and historical.

The effects of redundancy in language are antagonistic in two ways:

- reduction of the readability and comprehension by making the text longer and more complicated (this is the case for example with periphrases): *to make clear by avoiding redundancies*,
- on the contrary, reduction of the comprehension errors due to phonetic changes, due to noise made by the environment, due to a receptor's lack of knowledge: *to make clear by repetition*.

To conclude, the unnecessary *elements* introduced in the definition of the term *redundancy* are syntactic parts or semantic information of the sentences. A text is a structure which is scanned by the reader who deduces a meaning. Scanning and meaning define the input/output relationships. If the meaning of the text is correctly understood, these relationships are qualified as *normal*. Some text elements are redundant if their removal does not modify the correct meaning. The repetition of lexicographic or syntactic *elements* in the sentences is not necessary if the comprehension is good. On the contrary, these elements are useful in case of a failure of comprehension.

For instance, consider a reader of the previous syllogism who has not made the semantic correlation between the first two phrases and the last one. Therefore, he/she has not concluded that Socrates is mortal. This deduction is therefore brought by the third phrase, which is hence found to be useful for the comprehension. In this example, illustrated by *Figure 8.1*, we see that redundancy is a useful tool for increasing the quality of the comprehension of the semantics of the sentences, but that it is difficult to master.

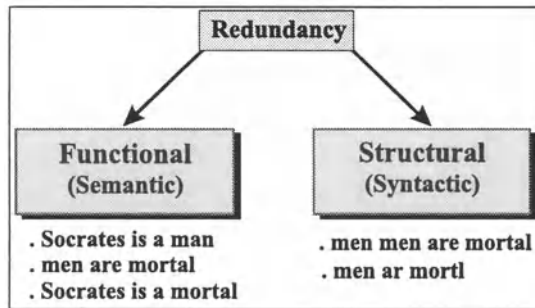


Figure 8.1. The two aspects of redundancy

8.1.2 Redundancy of Computer Systems

In the context of design, production and use of electronic products, we will also often meet difficulties when trying to master the redundancy and distinguish its positive and negative aspects relatively to the dependability requirements. The redundancy of electronic system also presents two forms (*Figure 8.2*):

- *functional redundancy* which corresponds to *semantic redundancy* in the case of linguistics,
- *structural redundancy* which corresponds to *syntactic redundancy* in the case of linguistics.

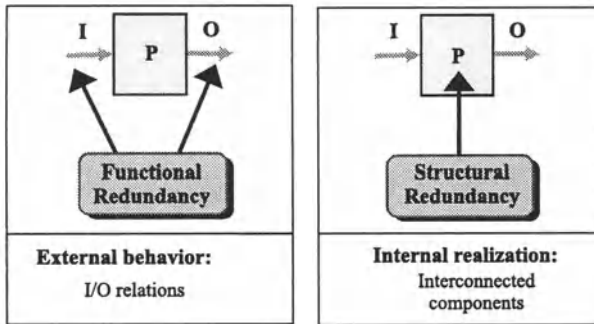


Figure 8.2. Functional and structural redundancy

The *functional redundancy* of a product is a characteristic of its *external* behavior in its functional environment: certain input values or sequences are never applied whilst the product could react, or certain output values and sequences are never produced by the product during its functioning. A simple example is that of a system adding two 1-digit decimal numbers. The result obtained is a two-digit number, but only the configurations between 00 and 18 are possible: redundancy in this case concerns the output values (19, 20, ..., 99) which cannot appear in reality, whereas the dimension of the output would potentially permit them (see *Figure 8.3*).

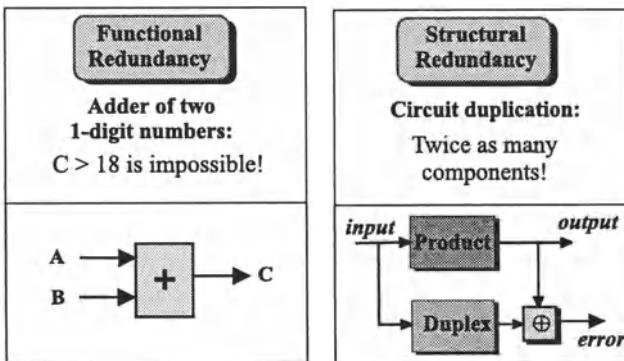


Figure 8.3. Simple examples of redundancy

This type of redundancy is also due to constraints between inputs and outputs: for example, if a sub-program calculates the greatest value of a list

of numbers provided as inputs, the property ‘the greatest value belongs to the initial list’ expresses a functional redundancy: not any value can be normally returned by the product; the output value is constrained by the input values.

Whereas the functional redundancy characterizes the external behavior of a product, the *structural redundancy* depends on its internal structure: there is more hardware or software than necessary. The duplication of two circuits with identical outputs is such an example. This is a duplex technique studied further on and illustrated in *Figure 8.3*. There are twice as many transistors or logical gates than are strictly necessary.

In the software domain, the definition of a constrain type such as ‘subtype Shoe_Size is integer range 28 .. 46;’ in a program also constitutes a structural redundancy. Indeed, only the type *integer* is necessary to generate the memory allocation of variables of this type as well as the arithmetic operation code (+, -, *, /). The constraint ‘range 28 .. 46’ will generate assembly instructions, which are useless where the function is concerned. If no fault is committed upstream the program, these instructions do not serve any purpose: they constitute a structural redundancy. However, such a redundancy is clearly useful in order to verify the type of values provided to or calculated by the program, and to detect possible errors.

These two types of redundancy are complementary: a product can present structural and functional redundancy at the same time. The two following sections develop these notions for hardware and software systems.

8.2 FUNCTIONAL REDUNDANCY

From a purely functional point of view, the product to design and then implement carries out a certain treatment of information provided by the functional environment via the inputs. This product treats this information, then it sends back the results transmitted to the environment by the outputs. Functional redundancy is going to qualify the product’s behavior relative to its inputs/outputs relationships.

A product has *functional redundancy* if:

- some theoretically possible *input values or sequences* are not applicable according to the product’s specifications,
- some theoretically possible *output values or sequences* are not produced according to the product’s specifications,
- some theoretically possible *input/output values or sequences* never occur according to the product’s specifications,

This definition conforms to the general definition of *redundancy* given in 8.1.1. It considers as *elements* the input and/or outputs values, and as *normal* situation a correct usage and/or functioning. This kind of redundancy, is independent of the product design and implementation, as it concerns the product function. The modeling tools allowing this redundancy to be characterized are studied in this section.

8.2.1 Static Functional Domains

Imagine that a product P has n inputs and m outputs whose values are expressed in any numeration base (binary, decimal or other) $B = \{0, \dots, b\}$. The values taken by the inputs (and the outputs respectively) are called **input vectors** (and **output vectors** respectively). We suppose that this product could be a combinational or a sequential system. The output values of a combinational system only depend on the applied input values, whereas the output values of a sequential system depend on the applied input values and the internal state which expresses the system's behavior by a finite state machine.

U_{SI} and U_{SO} are the *static universes* of all the possible theoretical input and output vectors. For example, a product which has $n = 3$ binary inputs and $m = 2$ binary outputs possesses:

- a *static input universe* with 8 vectors $U_{SI} = \{000, \dots, 111\}$,
- and a *static output universe* with 4 vectors $U_{SO} = \{00, 01, 10, 11\}$.

8.2.1.1 Static Functional Domain of Inputs and Outputs

The static behavior of the product introduces the notion of *static functional domain*. We call:

- **static input functional domain**, noted D_{SI} , the set of the vectors applied to the product by the environment, as defined by the specifications, that is to say without faults in the environment: $D_{SI} \subseteq U_{SI}$,
- **static output functional domain**, noted D_{SO} , the set of the product's output vectors which result from its activity, as defined by the specifications, that is to say without product failure: $D_{SO} \subseteq U_{SO}$.

A combinational system is characterized by a mathematical application from the D_{SI} domain in the D_{SO} domain, as illustrated by *Figure 8.4*. For each vector applied at the input, the system gives an output vector.

▮ A **static (input/output) functional domain is redundant** if and only if it is strictly included in its static universe.

This implies that certain vectors of the static universe are not part of the product's specifications defining the product relationships with the

functional environment. This is symbolized by the crowns in light gray in *Figure 8.4*. We define the **static (input/output) functional redundancy rate** as $(\text{size}(U_{Sx}) - \text{size}(D_{Sx})) / \text{size}(U_{Sx})$, where $x = I$ for inputs and $x = O$ for outputs.

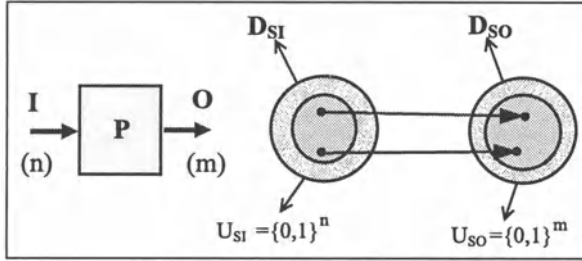


Figure 8.4. Static domains of a combinational circuit

Example 8.1. Decimal adder

Let us consider a decimal adder which receives two 1-digit numbers a and b and provides the result c on two decimal digits. Whether implemented in the form of a hardware or software system, this product presents a functional redundancy of the outputs, as illustrated by *Figure 8.5*.

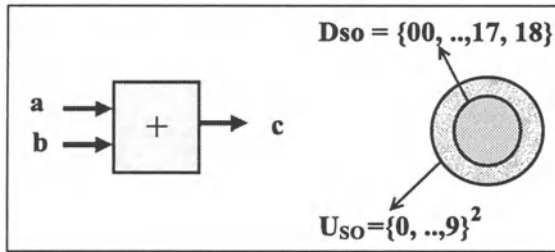


Figure 8.5. Decimal adder

Indeed, assuming that all the configurations applied to the input are possible, the static input domain does not present redundancy ($D_{SI} = U_{SI}$). On the contrary, c has two digits. Therefore, the output universe has 100 numbers whilst only 19 of them (the numbers between 0 and 18) will effectively be calculated by the product. Thus, there is an output redundancy of 81% of vectors!

Functional redundancy is an interesting concept. For instance, it allows the detection of failures that imply an output of the functional domain. In the case of the previous example, an observer placed at the output of the decimal adder can detect a failure by checking that the result effectively belongs to the output domain: $c \subseteq D_{SO}$. Thus, the result $c = 56$ is perceived as a failure.

Therefore an internal fault exists in the product, but the location of this error is unknown for the moment. This failure defines a class of equivalent faults from the external observation point of view (*Figure 8.6*).

In the third and fourth parts we will present several examples of functional redundancy applications, and show how this notion increases the system dependability, notably in the case of software.

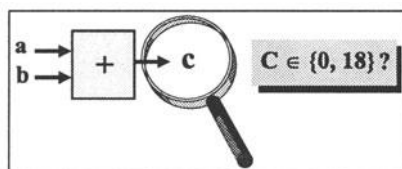


Figure 8.6. Observation of the adder

8.2.1.2 Static Functional Domain of Input/Output Relations

In the previous section, redundancy concerned either inputs or outputs. In a more general way, functional redundancies exist which correlate the input and output domains. Thus, we define a tuple of input and output universes, noted as $U_{SIO} = U_{SI} \times U_{SO}$. The input/output domain, noted as D_{SIO} , comprises the set of all possible vectors in U_{SIO} . A product has an *input/output functional redundancy* if $D_{SIO} \subset U_{SIO}$.

Example 8.2. Search for the greatest number of a list

Consider a system which receives 4 natural 1-digit numbers and which gives out the greatest number. The input universe has 10^4 vectors $\{0000, \dots, 9999\}$, and the output universe has 10 vectors $\{0, \dots, 9\}$. Consequently, the input/output universe contains 10^5 vectors. The input/output domain has only 10^4 vectors, because for each input vector the product only provides one single output value which is one of the entered numbers. Consequently, the input/output static functional redundancy rate is 90%, whereas no static redundancy is revealed (neither at the input nor at the output).

8.2.2 Dynamic Functional Domains

What we have just discussed regarding static function (in terms of vectors) is also relevant for the dynamic behavior of sequential systems. Now, we no longer deal with the input and output vectors but with the input and output *sequences* of vectors. We assume that all these sequences are of a finished length. Thus, we name the set of all these theoretical sequences which can be formed with the input and output sequences, the *dynamic input functional universe* (U_{DI}) and *dynamic output functional universe* (U_{DO}).

With the same generalization used for static domains, we define the *dynamic input/output product functional universe* by $U_{DIO} = U_{DI} \times U_{DO}$.

8.2.2.1 Dynamic Functional Domains of Inputs and Outputs

The *dynamic input functional domain*, noted as D_{DI} , is the set of the input sequences applied to the product by the environment, conformably to the specifications: $D_{DI} \subseteq U_{DI}$,

The *dynamic output functional domain*, noted as D_{DO} , is the set of the product's output sequences resulting from the product activity conformably to its specifications: $D_{DO} \subseteq U_{DO}$.

A sequential system is characterized by an application of the dynamic input domain in the dynamic output domain.

As for static domains, we say that a *dynamic input domain* or a *dynamic output domain is redundant* if and only if it is strictly included in its universe.

Example 8.3. Binary counter

Let us consider a 4-bit binary asynchronous counter. Each time it receives a pulse on its asynchronous I input, it increments a memorized value and sends it to the O output in the form of a 4-bit number. We note this operation: $O_{j+1} = O_j + 1$ [modulo 16]. Such a circuit is used for example to count the number of objects which cross a certain space. The analysis of static input and output domains shows that there is no static redundancy; indeed, all the input values are applied, and the counter can take any of the 16 output values. On the contrary, if we determine that the dynamic output domain has sequences of length 2, we obtain the couples (O_j, O_{j+1}) modulo 16, that is 16 different couples, whilst the dynamic output universe for length 2 sequences has a cardinality of $16 \times 15 = 24$. Therefore, the redundancy rate is more than 93%.

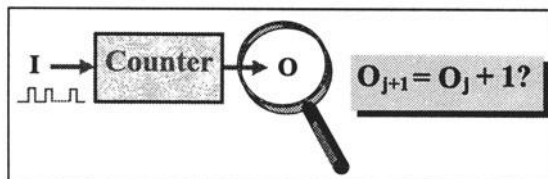


Figure 8.7. Observation of the counter

As in the combinational case of Example 8.1, we can use this redundancy by placing an observer at the product's output which memories the output's

two successive values and detects any dynamic domain violation (*Figure 8.7*). These failures are created by a class of faults which depends on the product's design. Hence, the functional redundancy is exploited to detect errors by means of an automatic observer which is itself redundant according to the normal functionality of the counter.

Example 8.4. Control software module

Let a sub-program, which, at each call, acts on an external process (for example an engine) by an output variable which takes alternatively the two values 'On' and 'Off'. These values provoke the running and stopping of the external process. If we consider the output sequences of length 2, the dynamic output universe is $\{(On, On), (On, Off), (Off, On), (Off, Off)\}$. However, each sub-program call aims at changing the state of the external process. It stops if it is the running and makes it run if it was stopped. Therefore, the dynamic output domain only comprises of two values: (On, Off) and (Off, On). There is a dynamic redundancy rate of 50%.

8.2.2.2 Dynamic Functional Domain of Input/Output Relations

We call *dynamic input/output functional universe*, noted as U_{DIO} , the set of sequences $U_{DI} \times U_{DO}$.

As for the static domains, we define the *dynamic input/output functional domain*, noted D_{DIO} , as the set of input and output sequences which are in conformance to the specifications.

We say that there is a *dynamic functional redundancy* if D_{DIO} is strictly included in U_{DIO} .

We consider Example 8.4 again. We suppose that, in addition, the sub-program named 'Control' switches off automatically the process after a delay D (it acts as a timer). This, for example, could be implemented in Ada language using a task and a variable of type 'duration' (which allows the time to be managed). The input universe has two values {Control sub-program call, D }. This last D value symbolizes the fact that a D duration has elapsed since the last call of Control. In this new context, the relations correlating the inputs and outputs lead to a dynamic input/output domain of length 2 that can take the values:

- $\{(Control, On), (Control, Off)\}, \{(Control, Off), (Control, On)\}$ if the interval between the two Control calls if inferior to the D duration,
- $\{(Control, On), (D, Off)\}$ in the opposite case.

8.2.3 Generalization of Functional Redundancy

To sum-up the previous explanations:

- | A product possesses a *static functional redundancy* if one of its static functional domains is redundant.
- | A product possesses a *dynamic functional redundancy* if one of its dynamic functional domains is redundant.

When a deterministic finite state machine describes the behavior of the system, the output value results from the input value and the current value of the internal state. The theory of languages shows the equivalence between certain expressions of input/output sequences (language) and the automaton model. Redundancy expresses itself in terms of states and/or arcs non-used by the functioning of the product: for example, it is impossible, from an initial state to lead the automaton describing the behavior into a state which is however part of its specifications. Example 8.5 illustrates this notion.

Another (and independent) extension of functional redundancy leads to a probabilistic vision of the domains. We end up with more general studies which belong to the domain of the *theory of information* and its applications in the detection of errors. Indeed, as the reader could ensure by treating Exercise 8.1, the adder’s output domain of Example 8.1 presents a non-uniform spectrum of the probability of occurrence of output vectors. This knowledge can be exploited in order to decide the likelihood of the frequency of appearance of a given vector; thus, the value $c = 9$ has to statistically appear 10 times more often than $c = 0$ or $c = 18$. The studies, which analyze these probabilistic aspects, will not be developed in this book.

Example 8.5. Redundancy of a FSM

Figure 8.8 shows an example of a redundant 4-state automaton. Let us suppose that the initial state is state 1, and that the inputs are constrained by the property ‘the input c is never applied after the input b’.

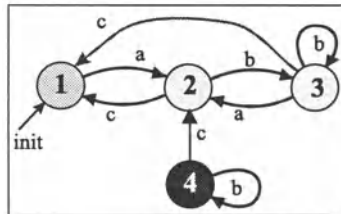


Figure 8.8. Redundancy at FSM level

With these conditions, it is easy to show that the arc from state 3 to state

1 is redundant, as it will never be used. State 4 is also redundant, as it cannot be reached from state 1: it is called an *unreachable state*.

A variation of this FSM will be studied in Exercise 8.2.

8.2.4 Redundancy and Module Composition

The design process, which structures a system into interconnected modules, frequently introduces functional redundancy because of the constraints due to the relationships between the modules.

The general problem, not developed in this book, can be expressed on the structure of *Figure 8.9* which comprises a father module M_{12} structured into two interconnected son modules, M_1 et M_2 . We know the global function of the father and the structure of the interconnection between the two sons. We suppose as well that the module M_1 is reused from another application. The module M_1 can present redundancy due to:

- the constraints of use of module M_{12} which are more restrictive than those predicted during the design of M_1 : all value sequences admissible by the M_1 are not applied to the external input I_e of M_{12} ,
- the constraints on the value sequences stemming from M_2 : all the internal input value I_1 sequences admissible by M_2 are not produced by M_1 .

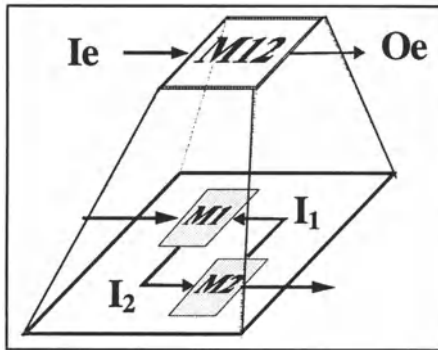


Figure 8.9. Reusability

The functional redundancy of structured systems can be formally analyzed using two operators: the *fusion* which determines the father’s functioning by the composition of the son’s functioning and the *emergence* which searches for the son’s function used at the father’s level.

- The *fusion operator* combines the behavior of the two son modules: $M_{12}^e = M_1 c M_2$, the operator c being the composition operator and M_{12}^e being the effective functioning of M_{12} resulting from the interactions between M_1 and M_2 .

The *emergence operator* determines the behavioral part of M_1 which is actually used in combination with M_2 to produce the behavior of M_{12} : $M_1^e = Em(M_1, M_{12})$.

There is a functional redundancy with regard to the module M_1 if: $M_1 > Em(M_1, M_{12})$ and/or if $(M_1 c M_2) > M_{12}$.

8.3 STRUCTURAL REDUNDANCY

8.3.1 Definition and Illustration

Independently of the functional redundancy associated with the specification of a product, the design stages which provide a structured system can introduce another type of redundancy: the *structural redundancy*.

A system presents a *structural redundancy* if its structure possesses certain elements which are not necessary to the obtaining of a behavior conform to the specifications, assuming that all the structure elements have a correct functioning.

For instance, structural redundancy of the implementation model corresponds to an overabundance of the resources used, in terms of:

- *hardware* (logical gates, electronic components or integrated circuits),
- *software* (statements, functions, procedures, data or objects),
- *time* (execution time of the algorithm and/or the circuit).

Whatever the system studied is, a theoretical design exists, sometimes inaccessible, which minimizes the resources used. Each additional element introduces structural redundancy, whatever the reason. This can be due to a non-optimal design, or even due to a voluntary duplication of the modules allowing the detection of errors, etc. Of course, the hardware aspects of this redundancy concern the physical components (for example electronic circuits). The software aspects concern the programming primitives (statements, variables), and the used software resources (operating system, etc.). Finally, the temporal aspects are relative to the product's execution time, whether hardware or software technology. We should note that these temporal aspects could be observed externally to the product. However, we include them in the structural redundancy when they are induced by the implementation means (circuits, programs) and not by the functional environment.

In the following sub-sections, we distinguish several forms of structural redundancy:

- on the one hand, between *passive redundancy* and *active redundancy*,
- on the other hand, between *separable redundancy* and *non-separable redundancy*.

8.3.2 Active and Passive Redundancy

8.3.2.1 Definitions

Structural redundancy is essentially studied on system model using primitive elements: for example a set of electronic components (transistors MOS), or logical elements (gates, Flip-Flops), or code lines. Redundancy exists as soon as the number of these constituents is greater than the optimal value: greater number of MOS components or gates, greater number of statements or variables in a program. This is therefore a theoretical notion, which, in many cases, is really very difficult to evaluate: in particular, the optimal values are often inaccessible.

A product possesses *passive redundancy* if certain elements can be removed without modifying the product's behavior.

A product possesses *active redundancy* if the number of elements is greater than the optimal value without direct possibility to removing one of them.

An element is *irredundant* if its removal causes the system to be functionally different

This distinction between passive and active redundancy is fundamental regarding the consequences on the dependability in general, and on the test in particular. The two following sub-sections illustrate these two notions on systems implemented by means of logical gates or features of a programming language.

8.3.2.2 Redundancy at Gate Level

The following examples show that the notion of active or passive redundancy is fairly easy to understand in the case of combinational logic circuits. This is more difficult to present in the case of sequential logic circuits composed classically of a combinational part and storage elements implemented by Flip-Flops (noted FF in *Figure 8.10*). We should note that, even if the combinational part is irredundant, the complete sequential circuit might have redundancy because of feedback loops created by the Flip-Flops. Irredundancy of the combinational part is a necessary but non-sufficient condition.

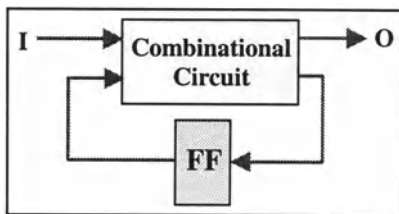


Figure 8.10. Sequential circuit

Concerning passive redundancy, the gates as well as their input/output wires are the elements considered. Before presenting examples of passive redundancy, the notion of **prime gate** is defined:

A gate of a logical circuit is said to be *prime* if none of its inputs can be removed without causing a functional change of the circuit.

Example 8.6. Passive redundancy: non-prime gate

The analysis of the function carried out by the circuit in *Figure 8.11* shows that the input noted as *X* of the OR gate is redundant and can be removed without changing the function *f*. Indeed, if the *X* input is present, the *f* function is: $f = a'.(a + b + c) = a'.b + a'.c$. (where *a'* is the logical complement of *a*). If the *X* input is removed, the function is the same: $f = a'.(b + c) = a'.b + a'.c$. This OR gate is therefore not prime, but it cannot however be totally removed. Thus, this example shows a redundancy of one input wire only.

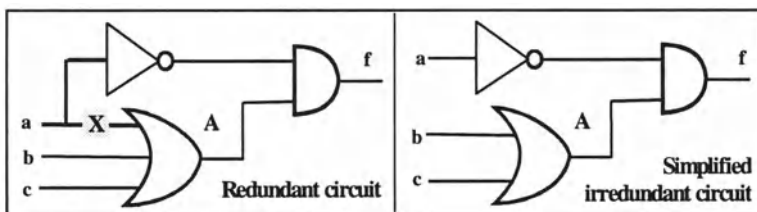


Figure 8.11. Redundant wire

Example 8.7. Passive redundancy: redundant gate

We take the simple example of a logical circuit with 3 inputs *a*, *b*, *c* and an output *f*, carried out with elementary gates: $f = a.b + a'.c + b.c$. A classical ‘SIGMA-PI’ realization of this function is shown by *Figure 8.12*: it has 3 AND gates and one OR gate. This circuit is redundant because the term *b,c* is useless in the *f* expression (a ‘consensus’ term derived from the first two terms). Indeed:

$$f = a.b + a'.c + b.c = a.b + a'.c + (a + a').b.c$$

$$= a.b + a'.c + a.b.c + a'.b.c = a.b + a'.c,$$

because the two last terms ($a.b.c$ and $a'.b.c$) are included in the first two ($a.b$ and $a'.c$).

Therefore, this circuit is not optimal. It presents a passive redundancy, as the gate $b.c$ can be removed.

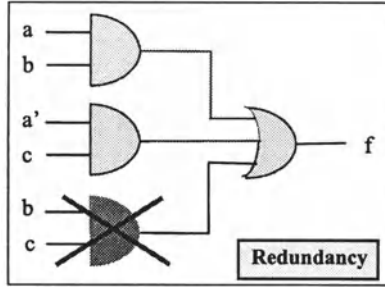


Figure 8.12. Passive redundant gate

Example 8.8. Active redundancy of a logical circuit

We now show an example of active redundancy obtained by adding a second output g on the previous circuit, such as $g = a.b + a.c$.

The realization shown by Figure 8.13 does not have passive redundancy, as we cannot remove one of the gates without changing function f or g . However, the term $a.b$, common to the two f and g outputs, could have been shared between these two outputs (the dark gray gates in Figure 8.13). This therefore is a case of active redundancy: all the elements are actively employed to produce the outputs.

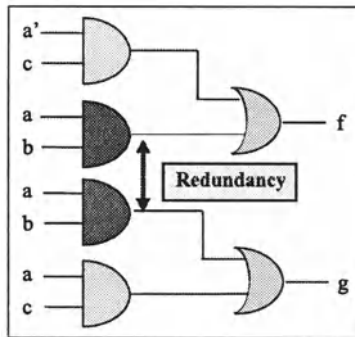


Figure 8.13. Active redundancy

8.3.2.3 Redundancy in Software Systems

The structural elements considered here are the statements.

Example 8.9. Passive redundancy of a program

We analyze the following extract of an Ada source program:

```
j := i;
k := i;
```

We suppose that the compiler has separately translated each of these statements into machine language by using an intermediate AX register. Thus we obtain the following assembly program:

```
mov AX, @i coding 'j := i'
mov @j, AX
mov AX, @i coding 'k := i'
mov @k, AX
```

The third instruction in assembly language, 'mov AX, @i', is redundant in a passive way, because AX already contains the variable *i*. This instruction can therefore be removed.

Example 8.10. Active redundancy of a program

We consider a program extract shown in *Figure 8.14* which computes the average value and then the sum of a set of floating numbers memorized in an array named Table.

```
Total := 0.0;
for I in Table'range loop
    Total := Total + Table(I);
end loop;
Number := Table'Last - Table'First + 1;
Total := Total / float(Number);
Put(" the average value is: ");
Put(Total);
Total := 0.0;
for I in Table'range loop
    Total := Total + Table(I);
end loop;
Put(("the sum of the values is: "));
Put(Total);
```

Figure 8.14. Redundant program

To provide the sum, the gray part repeats the calculation carried out to obtain the average. Thus, this program possesses a structural redundancy in terms of its code lines. However, we cannot simply pull out the gray lines, because the first accumulated value has been crushed by the average. This redundancy's suppression demands the program rewriting and the declaration of a second variable named *Average* in which the average value is assigned at line 6:

```
Average := Total / float(Number);
```

In a more subtle way, active redundancy is frequent when considering program's variables. Two variables of the same type can be used in different parts of a program whereas one would suffice. For example, we could find the following variables

```
The_Number_of_Registered_Passengers
and The_Number_of_Boarded_Passengers
```

in an airport management software.

The use of two distinct identifiers instead of one (*The_Number_of_Passengers*) renders the program more readable and also serves to detect errors (when their values are different).

8.3.2.4 Redundancy and Dependability

A first consequence of the notion of passive redundancy concerns the *detection* of faults. A redundant passive element may have faults which cannot be detected from the inputs/outputs of the system: we say therefore that this fault is *undetectable* or *masked*. *Figure 8.15* provides an example of such non-detection: the fault of the stuck-at '0' at the redundant gate's output cannot be detected by observing *f*. Other examples are given in the form of exercises at the end of the chapter.

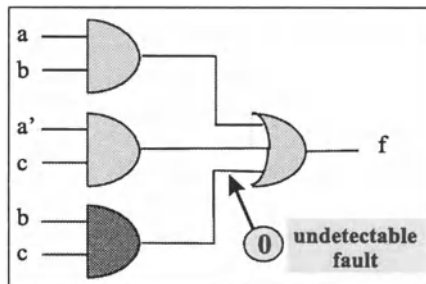


Figure 8.15. Passive redundancy: undetectable fault

Passive redundant elements (superfluous in functioning) could have been introduced involuntarily, for example because the method used had not

optimized the design. On the contrary, we will soon find other examples, which make use of passive or active redundancy in order to improve the dependability of products. Hence, we will introduce automatic detection mechanisms by duplication, or fault-tolerant structures by triplication.

In electronics, passive redundancy creates real problems for fault detection or diagnosis (*testing*) because of the ‘masking’ phenomena that have just been illustrated. We will find out about this problem later on, in the third part of this book. When dealing with ‘stuck-at’ faults models of gate arrays, the presence of a passive redundant element implies the existence of such non-detectable faults; on the contrary, in the case of active redundancy, all stuck-at faults can be observed as circuit failures for at least one of the applied input vectors: in that case, the circuit is said to be *totally testable*.

8.3.3 Separable Redundancy

In this sub-section we introduce another criterion to characterize structural redundancy. A product presents a *separable structural redundancy* if the redundant and non-redundant elements belong to distinct modules in the product’s structure. Therefore, we are talking about:

- *functional modules* which refer to the modules containing the functional elements,
- and *redundant modules* which refer to the modules containing the redundancies.

On the contrary, this redundancy is qualified as *non-separable* if it is not possible to separate functional and redundant elements into distinct modules. The redundancy is thus integrated into the original functional modules.

Separable redundancy is typical of duplication and triplication techniques. Each redundant module is also called *version* or *replicate* module.

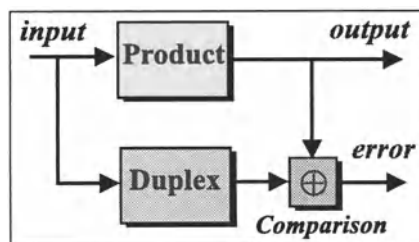


Figure 8.16. Separable redundancy: the duplex

Figure 8.16 shows a *redundant separable structure* called a *duplex*. This technique will be explained in the fourth part. The functional module is duplicated and the two module's outputs are then compared. An error is signaled as soon as the results given by the functional module and its duplex are different.

A criterion is often used to characterize separable redundancy:

- *on-line separable redundancy* (or *hot standby*),
- and *off-line separable redundancy* (or *cold standby*).

A redundant module is said to be *on-line* or *hot standby* if it is active at the same time as the functional module. This is the case of the previous example. The duplex is powered, and it receives the inputs and elaborates the outputs in parallel with the functional module connected to the external process. In Chapter 7, section 7.9, by using reliability block diagrams, we studied the reliability of 'parallel' settings which correspond to this type of redundancy.

On the contrary, a redundant module is said to be *off-line* or on *cold standby* if it does not function at the same time as the functional module. This module, called a *spare*, is only switched on when the primary module fails. Putting this spare module into service corresponds:

- to the electric power on and/or input/output connection to the environment for hardware implementation,
- to the execution or use of this module in the case of software implementation.

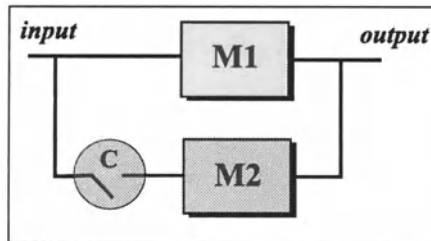


Figure 8.17. Off-line separable redundancy

Figure 8.17 represents a simple example of this redundancy: module *M1* is connected to the functional environment, whereas module *M2* is switched off, waiting to be activated. The *C* switch symbolizes this off-line waiting situation; it can also represent a switch of the redundant module power supply. It should be noted that in specialized papers the terms *active* for *on-line* redundancy and *passive* for *off-line* redundancy are often used.

8.3.4 Summary of the Various Redundancy Forms

Figure 8.18 sums-up the three main aspects of the structural redundancy:

- *hardware, software or temporal,*
- *passive or active,*
- *separable or non-separable.*

These criteria are independent. Numerous combinations of their values exist. For example, the duplex is a redundancy of active and separable type of hardware and software modules.

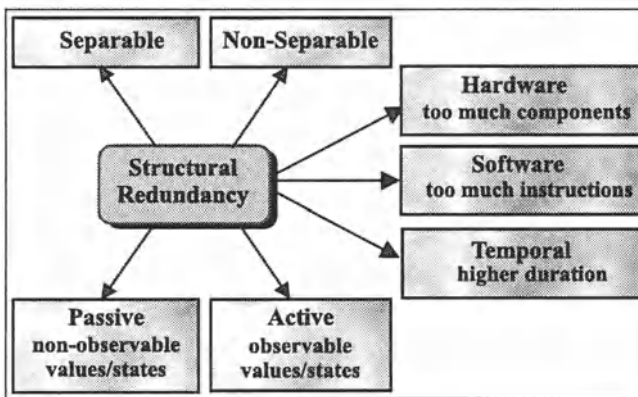


Figure 8.18. Structural redundancy

8.4 EXERCISES

Exercise 8.1. Functional redundancy of an adder

Consider the adder of Example 8.1 in section 8.2.1.1: $C = A + B$ ('+' is here the addition operator).

1. We suppose that inputs A and B are two one-digit decimal numbers, and that all (A, B) combinations have the same occurrence probability. We want to analyze the *probabilistic functional redundancy* of this circuit. Draw the probabilistic output functional domain and deduce the existing functional redundancy. How can this information be used to detect calculation errors?
2. The two numbers A and B are now in *Natural Binary Coded Decimal* (with 4 bits): $0 = (0000)$, $1 = (0001)$, ..., $9 = (1001)$. Knowing that the inputs have the same probability, determine the input and the output

functional redundancy rates of this system.

- We suppose now that the two input numbers are binary with two bits, and that an external constraint exists between these numbers: $A \leq B$. Determine the input functional redundancy of this product.

Exercise 8.2. Functional redundancy of a state graph

The state graph of Example 8.5 is modified as shown in *Figure 8.19*. We assume the same hypotheses as in Example 8.5:

- the initial state is state 1,
- the inputs are constrained by the property ‘the input c is never applied after the input b ’.

Analyze this graph and determine its functional redundancies.

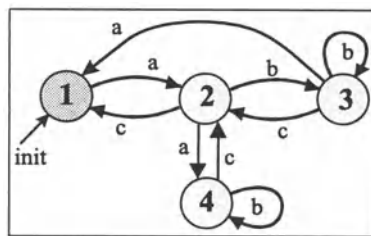


Figure 8.19. Redundant FSM

Exercise 8.3. Structural redundancy and faults

Consider the circuit of *Figure 8.20* which has two inputs and two outputs. Let us suppose that this circuit can be affected by stuck-at ‘0’ or ‘1’ of the wires noted α and β .

- Study the failures induced by each of these faults.
- From this study, deduce structural redundancies.
- Does this circuit present functional redundancy?

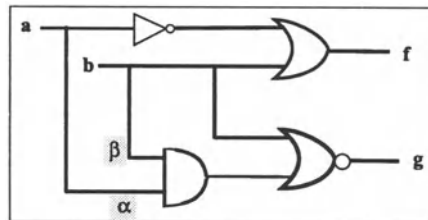


Figure 8.20. Redundancy of a circuit

Exercise 8.4. Structural redundancy of several circuits

1. Determine if each of the circuits in Figure 8.21, has passive and/or active structural redundancies.
2. Work out the logical structures of the corresponding non-redundant circuits.

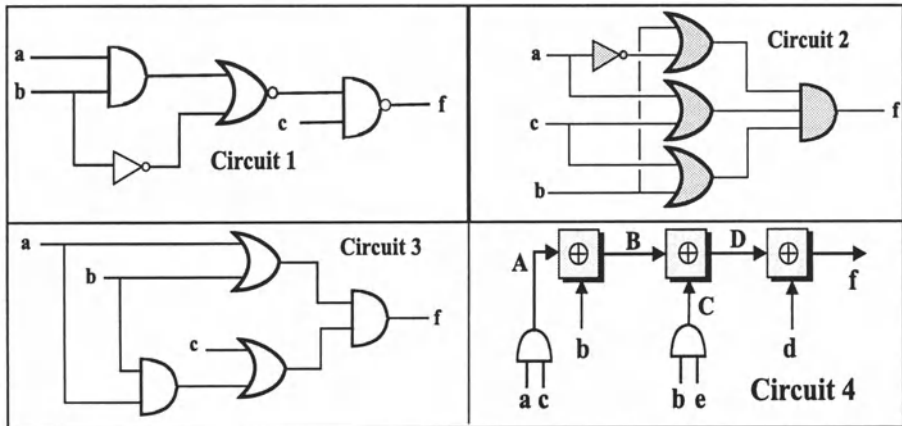


Figure 8.21. Structural redundancy of several circuits

Exercise 8.5. Software redundancy and constraint types

We consider the following statements:

```
subtype Size_of_Shoe is new integer range 28..45;
P: Shoe_Size;
```

1. Does the declaration of a new type (new) instead of using the type integer (P: integer) introduce a redundancy? If yes, is it functional or structural? Is it active or passive?
2. Refer to the previous questions for the statement of the constraint: 'range 28..45'.

Exercise 8.6. Exception mechanisms of languages: termination model

Programming languages such as Ada offer mechanisms which permit the detection of error occurrence and provoke the call to an exception handler which terminates the interrupted execution.

Example:

```
procedure XYZ(. . .) is
  -- declarative part
begin
  -- current body
exception
```

```
when others => -- exception handling
              --
end XYZ;
```

Analyze the redundancy characteristics due to the exception mechanism. Is this redundancy: active or passive, separable or non-separable, on-line or off-line?

Chapter 9

Avoidance of Functional Faults During Specification

9.1 INTRODUCTION

9.1.1 Specification Phase

The use of a product is fundamentally justified by the user's needs. The user possesses the initial motivation to buy or develop a product. In certain cases, this motivation corresponds to a necessity. For example, the fact that society does not accept accidents caused by the simultaneous presence of a train and a vehicle on a railroad crossing, justifies the creation of a system that avoids such accidents. Therefore, a product's life has to naturally start with the client's or future user's *requirements* (also called *needs*).

Then, this life cycle carries on with the product's *specification* stage in response to the previous needs. The previous example shows that one *need* could involve radically different specifications: a level crossing, or a bridge or a tunnel. The result of this stage is called *specifications*.

From the product's specifications, we obtain the *system* by a descending process known as *design*. This concerns a succession of stages, which are going to structure a system using the specifications expressed at an abstract level, to result in a system which is finally materialized as a physical (electronic) product or a software implemented on a physical support.

This chapter focuses on the requirement and specification stages which are at the origin of numerous faults. Their avoidance is fundamental, as their detection during the design or production stages is generally very costly.

In Chapter 6, we introduced two approaches which permit fault avoidance, that is to say fault prevention and fault removal. The objectives of such means have been presented concerning faults which can happen

during the creation stages. In this chapter, we present the practical techniques to reach the objectives assigned to these means during the requirement and specification stages. The integration of fault prevention and fault removal techniques in the same chapter is justified by their simultaneous use during the studied stages and their close correlations. *Figure 9.1* shows the location of these techniques in the life cycle.

Fault avoidance techniques used during design are considered in Chapter 10. The mastering of faults associated with the technology used (electronic or software) to implement the product is discussed in Chapter 11 (fault prevention) and Chapter 12 (fault removal).

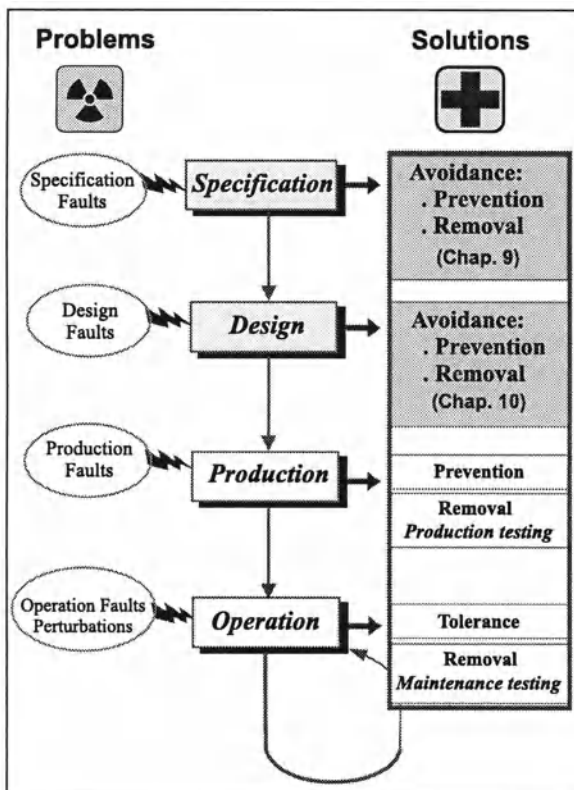


Figure 9.1. Fault avoidance during the specification and the design

9.1.2 Validation and Verification

The primary vocation of *fault avoidance* means is to prevent fault introduction during the considered stages of a system's creation process (here the expression of the requirements and of the specifications). During creation stages, we have seen that the faults introduced were due to the

method used to create the system (bad method and/or bad use of this method). Therefore, we seek first of all to master this method. Secondly, fault avoidance sets out to remove the faults introduced despite the precautions implied by fault prevention. We then need to identify the existing faults at each stage, in order to correct them. To do this, the result (what has been produced) of the considered stage is analyzed.

Fault prevention techniques can be divided into two classes:

- techniques acting on the *method* used during the considered stage,
- techniques acting on the *result* or *solution* of the considered stage.

The techniques of the first class allow a product to be developed in a correct way and to detect the erroneous aspects of the process used. Hence, we will speak of *validation of the method*. For example, these means seek i) to limit the incertitude of the method, in order to eliminate potential interpretation faults, ii) to limit the bad use of the method, in order to avoid the faults associated with its use. The techniques of the second class help the engineer to evaluate if the actual state of the developed product is correct. These techniques involve what we call *verification of the solution*.

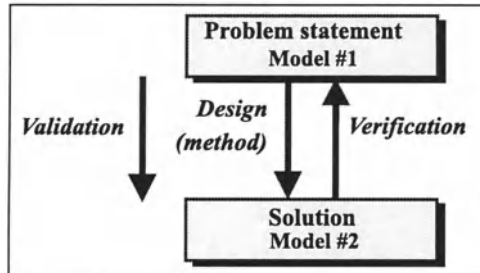


Figure 9.2. Design-Validation-Verification

Figure 9.2 synthesizes these two aspects. The statement of the problem as well as the formulated solution depends on the stage in consideration. For example, in the case of the specification phase, the statement is the requirements whilst the solution is a specification model.

We have to say that the two terms ‘validation’ and ‘verification’ often have a very similar meaning in current language. Opposed meanings to those proposed could even be envisaged. However, in this book, we will consider the meaning that has just been provided.

In the next part of this chapter, we present validation and verification techniques associated with the stages of requirement expression (section 9.2) and specification (section 9.3). The most popular verification technique, the *review*, is explained in section 9.4.

9.2 FAULT PREVENTION DURING THE REQUIREMENT EXPRESSION

9.2.1 Introduction

The system's specifications, that is to say the definition of a product to be developed, are derived from the client's or future user's requirements. Determination and expression of needs is difficult. This is probably one of the reasons for which we talk of 'need capture' techniques. The faults introduced in the expression of needs have two principal causes:

- a bad interpretation of real needs,
- a bad expression of needs which have been correctly understood.

In sub-section 9.2.2 we introduce a method which should reduce the presence of the first type of faults. In sub-section 9.2.3, we then propose a method in order to limit the occurrence of the second type of faults.

We present a simple and specific method for each of these two cases. Numerous other methods exist. Unfortunately, we cannot go into greater detail, as this would mean a book of an unacceptable length and a lack of generality. Moreover, our goal is to focus on dependability issues. So, in sub-section 9.2.4 we provide a way to evaluate the capability of methods to produce correct expression of needs. The reader can use this to judge the numerous other need expression's methods available, so that his/her choice is guided by the required dependability of the product to be developed.

9.2.2 Help in the Capturing of Needs

The needs are generally obtained by interviewing the client. To be efficient, that is to say to produce non-erroneous needs, these interviews have to be carried out with a certain method. One possible method consists in leading the interviews by seeking the responses to five questions: What? Where? When? Who? Why? The responses to each of these questions permits classes of generic faults to be avoided, that is to say those which are non-specific in the development of a certain product.

1. What?

We seek to define the entities upon which the future product has to act, as the needs concern the user environment. This allows the detection and exclusion of the elements which do not have relationships with the future product. In effect, the client often has worries or other needs in mind which he/she exposes but which should not interfere with the project; if not, they lead to the analysis of erroneous needs.

2. Where?

The objective of this question is to determine the localization of the elements brought out by the previous question. The answers split the entities into external entities (for example an electric signal which the product should take into account), and internal entities (for example memorized data which qualifies the product's state). This question is fundamental as it permits the separation of what should be inside the system and what should already belong to its external environment. This issue becomes increasingly critical in electronic systems due to their interaction with other systems (electronic, mechanical, human, etc.). Therefore, it is essential to define the location of enumerated entities in order to avoid starting a development assuming, for example, that a piece of information is delivered to the system by its environment whereas it should be computed by the system.

3. When?

This question aims at taking the temporal aspects into account. Where the product's internal entities are concerned, the response permits the expression of the state's sequencing. For the external entities, it defines, for example, the sequencing of events which could arise and the actions which need to be performed. For example, if a client desires a system which controls the access to a protected room, the sequence 'card entered - code captured - code correct - lock open' describes such a relationship between the entities.

4. Who?

The response to this question defines the actors which act on the entities and the actors influenced by them. This could involve the external actors (for example, the code is typed by the user) or the internal actors (for example, the code's validity is evaluated by the system). It permits the description of the agents who influence or who are influenced by the entities treated.

5. Why?

The client has to justify the necessity of the elements expressed. The response to this question allows, in particular, information to be perceived which is relative to the client's current preoccupations but which are not related to the project. This question is therefore redundant with the others. However, it permits each requirement to be analyzed again and verified.

9.2.3 Expression Aid

After having examined the first part of the method which concerns the capture of needs, we should consider the second part which helps the

expression of these needs. The availability of guides for such an expression is in effect indispensable, as many faults arise during the expression process. They are principally due to the high volume of information provided by the client. The mastering of this great amount of information is carried out according to two complementary approaches:

- the definition of entity families,
- the definition of abstraction levels.

The aim of the definition of the families of entities is to group the elements of information into classes of the same nature. For example, if an aspect of a problem concerns the scheduling of manufacturing activities, all the information relative to this subject have to be regrouped. Indeed, the client generally provides information in a disorderly way, that is to say without structure. He or she passes from one aspect to another of the needs as soon as an idea comes to mind, or when he/she remembers that a certain aspect has been omitted. This phase provides a *horizontal structuration* of the information, as represented by *Figure 9.3 a)*.

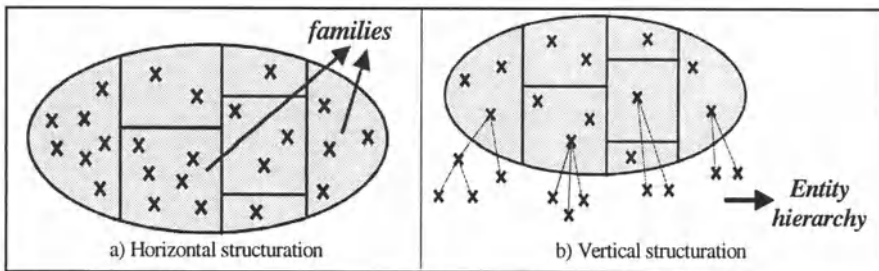


Figure 9.3. Information structuration

The aim of the second approach is to organize the elements of a class by detecting the hierarchical relationships linking them. For example, the development of a bank management system requires the following pieces of information belonging to a same class: bank account, client's name, bank balance, account holder's address. They can, however, be organized in the following hierarchical way. The bank account notion is more abstract: it provides a reference to a client and his/her account: the client himself/herself is defined by his/her name, his/her address, etc. The detection of the levels of abstraction permits a *vertical structuration* of the information in each of the classes obtained by the previous horizontal structuration into families. *Figure 9.3 b)* represents such a vertical structuration.

These two methods of information organization prevent faults such as the use of two different terms for the same notion. Indeed, these two terms can be found in the same family where their equivalent meaning will be easier to

perceive. In the same way, the natural language's semantics being ambiguous, the hierarchy permits the clarification of the belonging links and therefore the fault detection through a comprehension of these links. For example, is a client's address an element of a bank account or does this information belong to two families if a client has several accounts?

9.2.4 Evaluation of a Method

We have just described a simple method for capturing and expressing needs. Many other methods exist, and to present them all would require another complete book. Whatever the choice, the chosen method has itself to be judged. This involves, therefore, a validation process which aims at responding to this question: am I developing a product in the *correct way*? The incorrect way could result from a bad usage of a good method (this will be detected in the resulting product), or a correct usage of a bad method. We are going to study the last point. It should not be forgotten that the underlying idea of this analysis is that a bad method will undoubtedly cause an erroneous model of the product. This method evaluation has to be done using criteria. Where the expression of needs is concerned, we use the 8 criteria described below.

1. Facilitate the comprehension of the method

It is clear that a method can be effective in theory, but that in practice it can be applied in an erroneous way (and therefore produce a faulty result) due to the fact that it was badly understood by its user. Therefore, the relative simplicity of the method of need expression and capture is a first criterion of its efficiency for dependability purpose. This is the case of the method presented in the previous section.

2. Systematize its application

The method has to clearly guide its user in order that he or she cannot badly use it, but also so that he or she concentrates on the client's needs and not on the method itself (if it necessitates further analyses for its use). In particular, it has to clearly separate the explicit different stages and define, if necessary, their sequencing. The five questions regarding the capture method and the two classifications of information received by the expression method, which have been previously presented, have this goal.

3. Facilitate clarification of the problem

The aim here is to evaluate if the chosen method permits, or not, its user to separate the problem posed by other parasitic information. We seek to determine if the information retained is pertinent or not. In the previous

method, the majority of the questions and the need to class the obtained information concern, among others, this objective.

4. Facilitate clarification of the incoherence in the information provided by the client

The method could be *redundant*, that is led to ask the same information in several ways or possess the means to find relationships between pieces of information. This is typically an objective of the fifth question of the capture method proposed. The fact of asking why the client has a need does not provide additional information necessary for the establishing of the future specifications. The demand for a justification, however, allows detecting useless pieces of information, or consistency problems.

5. Facilitate the communication with the client

The analysis of the needs, and then the definition of the system's specifications, are the only stages in the development process which permit a dialogue with the client as well as the expression of an acceptance or disagreement by the client. Only the system's delivery will then be the object of such an exchange. However, this last stage is too late to state a disagreement. The method used to understand the client's needs therefore has to privilege the dialogue with the client.

6. Encourage the creation of documents

Documentation is an essential means of communication with the client. Writing documents also provides a way for the designer to avoid faults by obliging him/she to materialize (by a text) and thus analyze his/her understanding of the needs.

7. Take the changes into account with ease

Whilst a method has to be rigorous and systematic, it also has to be able to take modifications of the client's information into account. Indeed, the client can take advantage of successive interviews to define (and therefore modify or precise) his/her own needs.

8. Provide means favoring the partitioning, abstraction and projection of information

The five questions of the proposed method provide up to 5 projections of information. The classifications permit the *partitioning* (by horizontal structuring) and the *abstraction* (by vertical structuring).

Note. The information redundancy, which appeared in the previous elements, can be perceived as harmful as they increase the work of the need

expression and could lead to the creation of more faults. In reality, it aims at mastering need expression and fault detection by stating inconsistencies. A compromise between ‘not enough’ and ‘too much’ information is however difficult to find.

Being always expressed by informal languages (for instance, English), the requirements cannot be checked by automatic tools. Fault detection is led by human appraisals. The most important approach is the *review technique* which is useful to analyze the requirements as well as the specifications; this approach is introduced in the next section, and developed in section 9.4.

9.3 FAULT AVOIDANCE DURING THE SPECIFICATION PHASE

Once the client’s needs have been obtained, the engineer has to define the system which he/she wants to propose to answer these needs. This work leads to the *expression of specifications*. Here also, the faults relative to the specification stage are due to three causes: a bad understanding of the client’s or future user’s needs, the proposal of a system which does not respond to the needs which have however been clearly understood, and a bad expression of a specification which has been well thought out.

In order to avoid these faults, the engineers need adequate methods to help them in their work. This work is approached in sub-section 9.3.1 for the fault avoidance during the specification expression phase. Then, means permitting the product’s specifications to be evaluated in order to remove the faults are required. This is the object of sub-section 9.3.2.

9.3.1 Fault Prevention: Valid Method

9.3.1.1 Choice of the Modeling Tool

The work carried out during the specification phase consists in producing a model which defines the product to be developed. This model is expressed using a *language* (or *modeling tool* or even *model* by language abuse). This work will be facilitated, and therefore the number of faults will be reduced, if the chosen modeling tool offers features close to the concepts of the system’s domain. For example, if a system’s behavior to be specified is purely sequential, the use of a ‘finite state machine’ model is well adapted. On the contrary, if it is necessary to represent parallel activities, the use of a model such as Petri nets is preferable. The objective of the dependability implies thus firstly *an explicit and justified choice of a modeling tool*.

It is here difficult to provide an exhaustive enumeration of such means of modeling and criteria which define their suitability to application domains.

Indeed, for each specific problem, it is necessary to determine its domain and then choose the means of modeling. We should also note that this choice problem is not particular to the specification model but will also be necessary for the design models and the implementation models. For this reason, we will tackle the selection technique for treating design in Chapter 10. We should only insist here on the importance of this choice on the dependability of the systems produced. A non-adapted modeling tool will without doubt lead to a complex modeling which renders the understanding difficult and thus will create faults during the following phases.

Even if the modeling means is specific to each application domain, it has to possess intrinsic qualities. For instance, it has to compensate for the intellectual limits of all human beings, including the specification team members. In particular, it has to allow abstraction expression in order to obtain more or less detailed views to facilitate expression and understanding. These permit the use of a limited number of abstract objects whose reality will be defined in the ultimate stages. On the other hand, it has to have a precise semantic so that a feature cannot be interpreted in different ways by the system designers. The question therefore concerns the use of a formal model. We make two remarks regarding this point:

- it is important that the models described from such a modeling means are understandable by the client so that he/she can give his/her approval to the definition of the specified product,
- it is essential that the modeling means used permit the expression of different views (or abstractions) of the defined system, such as the inputs/outputs, the behavior, etc., whose redundancies allow checking to be done.

9.3.1.2 Mastering of Modeling Process

The knowledge about a modeling tool is not sufficient. Still with the objective to develop a dependable system, the designer has to dispose of guides helping him/her to transform the client's needs into the system's specifications. He/she should *use these guides and demonstrate their real use*. Here again, the guides which can be proposed depend on the modeling tool used. For example, the reasons for using a given synchronization when using Petri nets as specification tool are relative to this model feature and to the classes of systems specified. In order to remain coherent about this book's objectives, we cannot provide a complete guide for a given model. However, examples of information that should be contained in this guide will be studied in Chapter 10 dealing with the design process, as an identical problem exists. Indeed, such guides have one general aim: to help the deriving of a new modeling from an existing modeling.

9.3.2 Fault Removal: Verification of the Specifications

The aspects described in the previous section are associated with the *validation* (how can a specification be produced in a correct way?). Once produced, the specification has to be verified in a way which brings out possible faults. This is the theme of this section.

9.3.2.1 Verification Parameters

Specification verification seeks to detect the presence of faults. To do this, there are two approaches:

- the first one studies the system's role by looking to confront with the user's needs,
- the second one carries an intrinsic judgment on the quality of the specifications expressed, without being preoccupied with what they express; the underlying idea is to think that an expression of bad quality has a high risk of containing faults.

1. Specific fault detection (or conformity)

Specification verification can seek to detect two types of faults:

- faults which provoke characteristic errors of the modeling tool used,
- faults specific to the modeled system.

For example, the Petri net model allows the detection of a deadlock between parallel cooperative activities. In the case of the use of a formal modeling means, an automatic tool can perform the detection of these errors which are characteristic of this modeling means. These characteristic errors are qualified as *generic*. Following this, a human expert has to diagnose the faults which are at the origin of these errors.

In the second case of faults specific to the modeled system, we will seek, for example, to show if the modeled system can reach non-desired states whose definition depends on the particular system considered. Thus, if a system controls the barrier of a level crossing, the state 'the train passes on the crossing and the barrier is open' is unacceptable. The client pays for the development of a system which guarantees that trains and cars cannot pass simultaneously (this is a need). Here also, the use of a formal model favors the use of tools and thus reinforces the guarantee that such undesired states will not occur, and therefore the faults which lead to it.

2. Qualitative verification

Specifications can be analyzed using certain qualitative criteria which are detailed afterwards. These criteria are generic faced with the specified

system and also with the modeling tool used. Here we judge the intrinsic qualities of the information contained in the specification document. These criteria therefore induce a third type of judgment on the modeling proposed.

The non-conformity to these criteria increases the risk of faults.

Consequently, contrary to the two previous evaluation types, this does not make obvious precise errors but only signals the potentiality of faults.

In order to illustrate this, we will give some criteria and sometimes advice in order to obtain the *conformity* to these criteria. Therefore, the fault risk will be reduced and thus many faults avoided. These criteria concern the semantic or the syntax of the proposed modeling.

Four criteria are generally associated with the semantics: *non-ambiguity*, *completeness*, *consistency* and *traceability*.

- ***Non-ambiguity.*** Each element of the specifications should only have one interpretation. One simple means to satisfy this criterion is the definition of a *glossary*, and the verification that all its words are used in the whole specification document according to the meaning given in the glossary.
- ***Completeness.*** We want to check that the specification predicts all possible cases. Any missing information can indeed lead the designer to substitute another information which is not desired.
- ***Consistency.*** We seek to establish that there are no conflicts between several specification elements. The glossary is again useful.
- ***Traceability.*** The engineer has to express the link between the client's needs and the system's specifications. A method which permits the introduction of justification of the model's constructs (here regarding specification) will be presented in Chapter 10, but the exposed method is applicable at every stage, including the specification.

Four criteria are associated with the syntactic aspects:

- ***concision:*** the specification statement should not contain useless verbiage,
- ***clarity:*** the statement has to be easy to read (it does not mean that its comprehension is easy),
- ***simplicity:*** the concepts manipulated have to be simple, in particular, the number of these concepts has to be limited and they should be loosely coupled,
- ***comprehension:*** the reading has to facilitate the understanding of the semantics.

Note. The previous criteria apply also to design models. Neglecting some specific features which will be signaled, we will find similar problems and

thus similar solutions.

9.3.2.2 Verification Methods

In sub-section 9.3.2.1, we pointed out that the use of formal models permits an automated analysis in order to detect the errors created by faults associated with the modeling tool and the faults specific to the modeled system (specific detection methods). When the model is 'executable' (e.g. a program or a model that can be simulated), this model can be considered as a product on which we can apply *dynamic verification techniques* which will be discussed in the following chapters (the *test techniques* principally). An analysis, generally performed by a human, has to then establish a diagnosis, that is to say find the faults at the origin of the error.

Where non-executable models are concerned, the analysis is human. Different methods exist: 1) *review*, 2) use of *scenarios*, 3) *prototyping*. We mention below the principles of these three methods.

1. Review

The *review* consists in a human analysis of the contents of the specifications. The reviewer may search for specific faults in the specification model. If we look for faults in a *qualitative* manner, that is searching for risks of faults, this review can only be carried out on a sample of these contents. In this case, we reckon that the violation of the criteria, if it happens on this sample, is without doubt repeated on the whole specification, as it is due to a bad work method. A team independent from the specification creation team can carry out this analysis, by studying the associated documents. Notes raising real or potential problems are transmitted to the specification creators so that they can provide justifications or act to take these remarks into account. The specification creators can also carry out the analysis during a presentation. The potential problems are therefore directly raised. The two approaches can be combined: an initial presentation reduces the study that is then refined in an isolated manner by the people leading the review. As the review techniques are very popular, they are presented in section 9.4.

2. Use of scenarios

From the specifications we derive input/output sequences which simulate the possible interactions between the environment and the specified system. These *scenarios* are then exposed to the client (or to the specification designer) who either approves or disapproves them. On the contrary, if the expression of need has already led to the expression of scenarios, they can be applied to the specified system. We find ourselves therefore in the *test* situation which will be discussed afterwards.

3. Prototyping

Prototyping consists in deriving a tool from the specification document. This tool simulates the system's interactions, but it does not correspond to a realization of the system. For example, the technological aspects of the future system (execution hardware, input/output devices, etc.) can be simulated in the prototype by software (data file, processing, etc) or by the operator (whose reactions substitute the absent elements). The tool's use by the client allows the detection of understanding and expression errors of his/her needs.

9.4 REVIEW TECHNIQUES

9.4.1 Principles

The *review* is a technique used to detect faults by analyzing the documents produced at the end of one or several phases. This technique is often used to examine the expressions of the requirements or of the specifications. Indeed, reviews do not need executable models. Hence, this allows partial and/or informal documents to be assessed. As a consequence, this technique can be used early in the development process, in order to detect faults as soon as possible.

The *reviewer* realizes four activities described hereafter.

1. He/she analyzes the current state of the system, but also of the process followed to obtain it. In particular, methods, techniques, and tools involved during the development stages can be judged, if the engineer produces documents specifying his/her way of working.
2. He/she expresses his/her conclusion concerning, sometimes existing faults, and often potential presence of faults. For instance, if global variables are used in a multitask application, a great risk of bad accesses to these shared resources exists. The reviewer has not to be sure of the occurrence of these problems: he/she has just to notice their potentiality
3. He/she communicates his/her conclusions to the authors of the analyzed documents, justifying his/her opinion.
4. He/she analyzes the reply and provides a final conclusion, as a set of actions to be done. Frequently, many elements expressed in the first opinion do not belong to the final proposed actions, as the authors have explained why the suspected problems cannot occur. For instance, if variables are shared by several tasks, the use of mechanisms guaranteeing their mutual exclusion for access, and the use of techniques proving the

absence of deadlock caused by these accesses, may not require additional action.

Sometimes, the reviewer also checks that the specified actions were actually applied.

In the following sub-sections, two techniques implementing the four previous activities are presented: *walkthrough* and *inspection*.

9.4.2 Walkthrough

Walkthrough consists in a presentation by the engineer (the author, also called the speaker) of his/her results (a system) and the process he/she used. During this talk, the reviewer asks for questions to improve his/her understanding and to express his/her opinion. The engineer answers and the actions to be done are defined immediately. Hence, the four previously mentioned activities are mixed together.

This technique has the advantage of training the reviewer. Thanks to the questions that are immediately answered, the reviewer obtains a good understanding of the system produced or the process. In particular, he/she must not read numerous documents. So, this review does not spend a lot of time, and therefore its cost is not prohibitive. For these reasons, this technique is often used as a first step of a fault removal process.

However, several drawbacks exist. Based on a discussion, this review process is not formal. The conclusions provided by the reviewer may greatly depend on his/her personality and the personal influence of the speaker. The review process may look like a bargaining whose results are hazardous. Finally, the speaker often masks pieces of information, intentionally or not. For instance, certain aspects are passed over in silence, as the speaker who has not spend enough time on some aspects of its work knows that problems may exist.

9.4.3 Inspection

Inspection is a review technique whose process is formalized by 9 steps.

1. **Request for inspection** by the developer, the client or an external authority. During this first step, a leader is chosen.
2. **Entry**. The leader establishes the review feasibility. In particular, he/she checks that all the useful documents are available.
3. **Planning**. The leader defines the inspection strategy (for instance, what are the critical aspects), the tasks to be done and their scheduling, and the persons who will carry out these tasks.

4. **Kick-off meeting.** The leader presents the objectives and ascertains that the selected experts understood their tasks.
5. **Individual analysis.** The experts search for actual faults or issues, which are expressed on special sheets described hereafter.
6. **Logging meeting.** The experts are grouped:
 - to enumerate the faults or issues noticed during their analysis,
 - to bring out other problems or to cancel certain issues, thanks to the knowledge of other experts,
 - to define additional studies to be made.
7. **Author answer.** The author of the part of the system for which an issue was signaled, answers. He/she may agree (if the fault actually exists) or disagree, justifying his/her reply.
8. **Actions to be done.** The group of experts analyzes the answers and decides if actions must be done.
9. **Checking.** The leader accepts or refuses the recommended actions, and then he/she checks their realization. An action advised by the experts can be rejected, due to the time of the money it requires. Therefore, the leader and his/her firm take the decision after he/she has considered that other actions have been applied to reduce the assumed risk, or that the potential failure is not dangerous.

The pieces of information relative to each issue are put together on a sheet which contains: a reference number, the expert name, the task identifier (for example, the part of the analyzed document), the description of the issue, the answer provided by the author, the conclusions of the group of experts, and the final decision of the leader concerning the realization of a recommended action.

Numerous criteria are used to analyze the documents during step 5 (*individual analysis*). They may be generic, by checking for instance if the process standards or the document writing standards specified by the project requirements were respected. For example, the capability of the requirement capture or specification method to help the engineer is assessed by these criteria. In this case, only samples of the documents are analyzed, as these criteria detect a bad method which was certainly used during all author's activity. Specific criteria are associated with a particular handled problem. For instance, if a specification document is reviewed, the reviewer checks that expected requirements are taken into account by the specifications.

The review is a non-automated process, as humans lead it. However, this approach is very efficient, as it highlight numerous erroneous situations due

to actual faults. Moreover, it does not require the execution of a formal model, so it can be applied on various documents and particularly during the first stages of the development process of a product.

9.5 EXERCISE

Exercise 9.1. Requirement analysis

The following text was recorded during an interview: “The communication means must be mobile. It must be transported in cars, ... It must have a maximized power autonomy, ... It must fit into one hand”.

Analyze this text to define the families of entities and their hierarchy.

Chapter 10

Avoidance of Functional Faults During Design

10.1 PRINCIPLES

Design is a complex stage of the creation of a product. It is potentially at the origin of numerous functional faults whose prevention and removal are quite difficult. The company *IBM* was the first industrial to publicly recognize the difficult nature of design faults. A fault analysis carried out on large operating systems of the 1970's revealed that not only did a small number of non-eliminated design faults exist, but also that the efforts made to totally eliminate them did not necessarily converge: eliminating a fault meant the appearance of other faults. This knowledge encouraged people to study and to use new methods and techniques, providing an important improvement of the product's dependability. As an example of the result of these efforts, the company *Fujitsu* announced in the 1990's that their software did not possess more than 10 faults on average per million program lines at the end of the first design. Even if these figures do not come from an independent organism, they are significant and should encourage the learning and use of the techniques introduced here.

The design stage is a top-down process transforming the specifications into a system. For example, the design of an integrated circuit successively produces the behavioral, functional, logical, electronic and technological models. Each model reveals new elements in relation to the previous level:

- from the *behavioral* to the *functional level*, we make appear the functions to perform and their relationships (functional modules are introduced),
- by passing to the logical *structural* level, we reveal the block primitives (gates, flip-flops, registers, arithmetic and logic units, memory, etc.),

- by passing to the *electronic* level, we use the commutation entities (transistors) and the electric power lines (which do not have any meaning in the previous levels),
- by passing to the *technological* level, the circuit becomes a topology with several technological layers: we encounter specific problems such as geometric dimensions and routing between elementary components (notably the power lines which have to be led to the processing places).

The same phenomena exist in the software domain where the *behavioral level* specifies the expected behavior of the future product (according to the HOOD and UML notations). This behavior is then expressed in the form of a structure of objects (using other features of the same notations), which are then transformed to the *software level* as a program written in a programming language. The technological level is often transparent, as the designer does not directly act either on the code generation or on the execution environment (use of compilers, input/output libraries, real-time kernel, etc.).

The development process of industrial projects is often more complex, as supplementary product integration stages exist at different levels. Let us give as examples, the integration of a software into a given hardware and software computing context, or even the integration of a module into an already developed product. Moreover, the sequencing of the step is not always linear (from behavioral level to technological level). Iterative design methods exist which progressively take the specification elements into account (incremental approaches). So, our splitting up of the design process into four phases offers a simplified view. However, in all cases, the transformations, which pass from one level to another, imply *methods* and their associated *models* at each level.

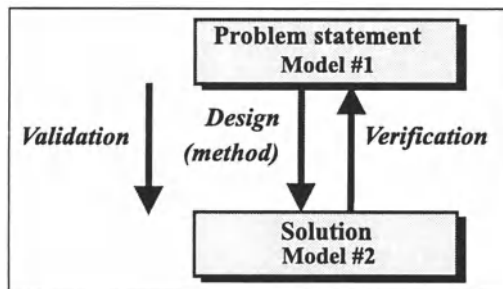


Figure 10.1. Design - validation - verification process

The means used to prevent or detect faults have the same objectives as those presented in the previous chapter for the expression of requirements or specification. Hence, the general scheme is identical (cf. *Figure 10.1*):

- use of an adequate method which permits a design model to be produced in a correct way (**validation**),
- use of means which allow the correction of the model produced to be ensured (**verification**).

The cycle ‘validation and verification’ is relevant for each phase of the design process. We obtain therefore the structure drawn in *Figure 10.2*: the design appears as a top-down chain of links, which, at each level, shows a model transformation which has to be validated, and the model produced which has to be verified. Thus, prevention and removal are closely coupled since the detected and corrected faults on one level are then prevented for the following levels. For example, the four levels represented in *Figure 10.2* can correspond to the four design levels of an integrated circuit previously mentioned. This chain translates the development process by successive structuration and refinement operations, from an abstract model until the final structure of primitive components. The model of the obtained system at a given stage makes the components appear whose specifications must be analyzed at the following level. Very often, due to reuse of already designed resources, this process is made more complex by adding a bottom-up process which assembles the available components to produce the complete system.

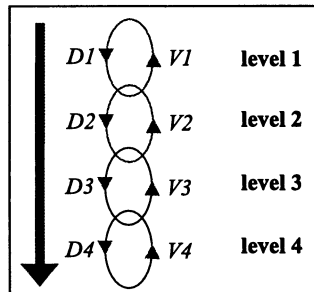


Figure 10.2. Multi-level design chain

Fault prevention is the first approach to consider. To put a validated design method into practice, three aspects must be considered:

- choice of a good method,
- correct application of this method,
- and checking that this application was correct.

The term *method* encompasses the *expression means* (also called *expression tool*, or *modeling tool*) of a design model, as well as the *development process* which allows a particular modeling to be obtained. The choice and the correct use of the modeling tool are studied in section 10.2,

and the design process is tackled in section 10.3. The use of a validated design method aims at preventing faults.

The second approach, *fault removal*, aims at verifying the model correction. It is discussed in section 10.4. Section 10.5 details one of the most important fault removal techniques: *functional testing*. Finally, section 10.6 proposes the study of some *formal proof techniques*.

10.2 PREVENTION BY DESIGN MODEL CHOICE

The general ideas introduced for the specification model remain pertinent for the design model. Here also, the choice of the expression means has to be carried out according to the characteristics of the system to be designed. Indeed, the more the model features are close to the concepts to be modeled, the more the design stage will be facilitated, and thus smaller the probability of introducing faults will be, as the solution will be less complex. It is difficult to develop this aspect without considering a particular application domain. We provide two examples to illustrate this idea.

- In order to express that a system is reacting to the occurrence of events, it is desirable to use a model integrating the notion of *task*. The asynchronous reaction (meaning here ‘in parallel with the current processing’) to the occurrence of an event could easily be done by specifying the reaction as a task and by associating this event with the task. On the contrary, the use of a sequential model necessitates studying when this event appears and inserting occurrence observation actions in the sequential activity, which makes the proposed solution more complex.
- If the problem is expressed using constraints which link pieces of data, then the use of the features offered by the CLP (Constraints Logical Program) is well adapted.

The choice of the design model can be carried out by showing the characteristics of the specified system to design, and by comparing them to the characteristics taken into account by the considered design approaches. This aim is discussed in the next section which tackles the design process.

The model has also to be capable of taking into account needs which are not associated with the designed system, but with the design process itself. We pointed out that this process is done by successive stages. The model used must allow abstract notions to be expressed whose realization (eventually partial) will be proposed at the following stage of design. If the model allows the expression of ‘abstract data types by identifying only their name and associated operations, it is possible for example to write at one design stage ‘the abstract type Stack offers the operations Push (in X) and

Pop (out Y)', and to manipulate objects of this type in the designed model. Then, in the following design stage, it is convenient to propose a realization of this Stack

The means proposed to answer to the needs relative to the nature of the system to be designed, and to those associated with the design process are sometimes coupled. It is the case with the notion of *Object* which is frequently used in software design. This facilitates first the expression of the application's entities (*class* notion), and then it permits them to be specialized (*heritage* notion). This answers the needs relative to the system's nature. For example, an 'acknowledge' window asking for a confirmation of a request by clicking on the buttons 'Yes' and 'No', inherits the properties of the generic class *Window*. This object notion is also a means associated with the design process. Indeed, it permits the manipulation of abstract entities by masking the means used during their implementation.

10.3 PREVENTION BY DESIGN PROCESS CHOICE

10.3.1 General Considerations

The choice of an adequate design modeling tool is necessary, but not sufficient for the design of a faultless system. Indeed, faults can also be due to the designer's difficulty in deducing a correct modeling thanks to this expression means. To do this, he/she disposes of a model at level *I* and a modeling means associated with level *I+1* (cf. *Figure 10.1* and *Figure 10.2*). Certain elements of the level *I+1* model can be automatically generated from the level *I* model. For example, if a level *I* model uses an abstract data type expressed by its specification, this can then be reproduced at level *I+1* where its implementation is defined. Thus, the specification becomes the interface of the designed component. Another example concerns the coding stage (writing of programs using statements). Numerous tools generate parts of the code, or skeletons of the code (Ada, C++, etc.), from the last design model.

However, the majority of the elements of a design model must be defined by the designer himself/herself. Without looking for automating this work (which would without doubt be in vain), there are two types of advice that can be provided in order to limit the introduction of faults, that is to say in order to create a correct model. The first class encompasses advice relative to the analysis process (*design guide*), whilst the second concerns the modeled system (*expression guide*). This last advice group being linked to the way a model is expressed, they are associated with the method, which justifies their presentation in this section. We are going to successively examine and illustrate these two aspects.

10.3.2 Design Guide

The model proposed at the end of a design stage has to combine features offered by the modeling means. For example, these features are *places*, *transitions* and *arcs* if we use Petri nets as modeling tool. These features are *loops*, *tests* and *sequences* if we use an algorithm or a sequential programming language as expression means.

The designer often meets the problem of knowing how to correctly combine these features in the aim of creating a correct model. It could be thought that the correction of this work (and therefore of the model) is uniquely a knowledge gained from experience. This is, in fact, the case today. However, we are going to see how this know-how can be formalized in the form of advice which can be used by everybody.

These guidelines depend, on the characteristics of modeling tools used or rather on the underlying concepts offered by these models (paradigm notion). We cannot give an exhaustive list of guidelines, as, on the one hand, we do not wish to treat a unique design model, and, on the other hand, this would again necessitate too many pages. We are uniquely going to give an idea of such guides by explaining their contents and contributions.

The person or team who proposes a modeling tool possesses an overall vision of systems world. Indeed, as a modeling is an abstraction form of a system, the features chosen to define a modeling tool are the elements considered as pertinent by the authors of this modeling tool. These features are supposed to be necessary for all modeling. Moreover, the authors have introduced these features in order to take into account the whole set of the specific characteristics of the domain of the considered systems. Remember the example of the '*Real-Time Systems*'. One of this domain's specific characteristics is the necessity of taking the occurrence of asynchronous events into account. To answer this need, design model creators have introduced the concept of *task*. This example shows two points:

- firstly, the models are adapted to a class of systems, and the choice of a particular model has to be carried out according to the belonging of the considered system to this class (this point has already been discussed in section 10.2),
- on the other hand, the introduction of modeling tool features has been deduced from needs associated with a system class, this is the second aspect which we develop hereafter.

If a designer knows the relationships existing between the features offered by a modeling tool and the needs associated with the domain of the considered systems, he/she possesses an essential aid to guide himself/herself in this design. It suffices that he/she *points out the needs* of his/her

particular problems in order to *deduce the means*, that is to say the features to be used. If, for example, the analysis of a system specification makes the existence of two events appear which have to create their own reactions during independent occurrences, two tasks *have* to be introduced in the design model. On the contrary, if a second event can only be taken into account after the reaction to the first event, one single task *has* to be used.

In conclusion, knowledge on the syntax and semantics of the modeling tool features is not sufficient for the designing of dependable systems. Knowledge of the feature origins is also indispensable. It is therefore desirable to dispose of guides which provide such information.

As well as the fact that these guides facilitate the deduction of a model, they also make corrections easier and provide the trace between the elements of the specification provided at each level I , and the elements of the design model proposed at level $I+1$.

10.3.3 Expression Guide

10.3.3.1 Principles

Whilst reading section 10.3.2, the experience acquired by the system's designers seems useless since the previous guides are only stemmed from the background of modeling means creators. This experience is on the contrary, very important, but in another domain, and its formalization leads to a second type of necessary guides. Indeed, despite the use of guides presented in the previous sub-section, the designed models can still contain faults. In particular, these are due to a bad comprehension of the system to be designed or a bad expression of a designed model which has otherwise been intelligently deduced.

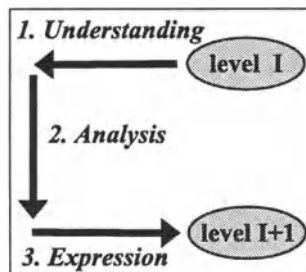


Figure 10.3. Phases of a design process

The design process uses three consecutive phases (Figure 10.3):

1. the *understanding* of the model of the system to be designed at level I ,
2. the *analysis* (deduction of the designed model),

3. the *expression* of the designed model at level $I+I$.

We should note that, once more, we find the three causes of faults relevant to the three aspects associated with the specification stage (see Chapter 9).

The first phase is the understanding of the previous level I model and the present state of the $I+I$ level modeling. Indeed, the modeling which stems from a stage is not produced in one single attempt but necessitates a certain period of time. Therefore, the designer has to make this design progress. Numerous faults stem from the difficulty which the designers meet when trying to master their own designed model during the whole design stage.

Then, an analysis phase is performed, which allows the modeling of level $I+I$ using level I data to be intelligently conceived. The guides proposed in the previous sub-section aim at avoiding faults associated with this phase.

We arrive finally at the expression of a designed model. Although the intellectual analysis is correct, numerous faults are due to bad expression of the correctly imagined solution.

The guides which we tackle in this part aim at avoiding faults associated with phases 1 and 3. Here also, the advices which can be given depend greatly on the modeling means used. In order to illustrate and put in concrete form the introduced notions, we consider the programming language as a modeling means. More precisely, in the two following sub-sections we provide some advice relative to Ada language. These guides can be adapted for other programming languages.

10.3.3.2 Understanding Improvement

Understanding is improved, and therefore the faults due to bad understanding are avoided, by firstly using rules relative to the *readability*. These concern for example, the following aspects of a programming language:

- 1) *lexicography*,
- 2) *self-documenting*,
- 3) *choice of kinds of words* according to the type of identifiers.

1. Lexicography

For example, we could quote as advice relevant to the *lexicography*, the use of identifiers constituted only of words whose first letter is in upper case and separated by the symbol ‘_’. Example: `Number_Of_Sold_Tickets`. On the contrary, the language’s reserved words are written in lower case. These rules highlight the words contained in the identifiers, that is to say the entities or concepts introduced by the designers. Indeed, we should suppose

that the language user knows well the language's features (if ... then ... else, while ..., etc.) which should then not be highlighted.

2. Self-documenting

Where *self-documenting* is concerned, we can quote three guides:

1. to pay attention to the meaning of the identifiers which have to be deduced from their reading,
2. not to use the constant values in the program body but to explicitly declare constant identifiers,
3. not to introduce comments to compensate for a loss of self-explanatory information.

Example 10.1. Bad use

```
for I in 1..320 loop
  -- we process the payments to social security
  -- of each employee
```

The *I* variable as well as the constants used in the loop (1 to 320) are not clear. In addition, comments have been added to try to remedy the situation.

Example 10.2. Good use

```
for Member_Of_Insurance_Company in Employees'range loop
  where the Employees'range defines the list of employees. If these are
  referenced by a number, we could have replaced this expression by
  First_Employee .. Last_Employee, where First_Employee
  (respectively Last_Employee) is an identifier of the constant which defines
  the first (respectively the last) employee.
```

3. Choice of words

Now, let us examine the guides which concern the *choice of words* according to the types of identifiers. For example, we advise the use of a verb as a procedure identifier to make the active aspect of this procedure explicit (execution of a processing when requested).

Other rules aim at *facilitating the understanding of the semantic*, for example, with the use of the *renaming* feature to define the specific meaning of a general notion in a particular context.

Example 10.3. Some rules

```
procedure Work_To_Be_Done(X : in Element)
  renames Stack.Push;
```

By this statement, each request for `Work_To_Be_Done` provokes a call to the subprogram `Stack.Push`. The calls to the procedure `Work_To_Be_Done(Y)` permits a better understanding of the action performed, while the call `Stack.Push(Y)` leaves a semantic fuzziness which risks creating interpretation faults.

Here we have only given some guidelines relative to the improvement of understanding to give the reader an idea of such rules and their contributions. Once again, an exhaustive list of guidelines would require an entire book.

10.3.3.3 Expression Improvement

The designer introduces numerous faults when the model is expressed. We will only quote one example, which is again illustrated at the programming level. This concerns a trivial case: *typing faults*. The proposed rules do not prevent these faults, but they facilitate their detection. Moreover, the compiler can automatically do most of these detections.

Let us consider the rule: 'Do not use constant values in the body of program entities (subprograms, packages, tasks, etc.) and explicitly declare these constants by identifiers'. According to this rule, the following program extract:

```
Hundred: constant integer := 100;
```

```
...
```

```
Rate := Value / Hundred;
```

is preferable to

```
Rate := Value / 100;
```

Indeed, if we type `Hunderd` instead of `Hundred`, this fault will be detected by the compiler, which is not the case when a keypressing error creates `1000` instead of `100`. We could retort two arguments:

First point. A keypressing fault could have taken place during the statement of the constant (`Hundred : integer constant := 100;`) where `10` was typed instead of `100`. This argument is true, but we estimate that:

1. This risk is smaller because the designer is concentrated on one single idea during this statement (the definition of a constant) whilst several elements intervene in the assignment of the expression: definition of an eventual complex expression, variable which will be assigned by the result.
2. The detection of a keypressing fault is easier in the declaration of the constant as the simple reading of `Hundred: integer constant := 10;` would make react a reader whilst he or she would perhaps not react to a statement like `Rate := Value / 10;`.

Second point. ‘The proposed guidelines impose an increase in the size of code, which is harmful to performance’. It is false in general. For example, the explicit statement of a constant does not have any impact on the code generated, as all the compilers substitute the constant’s value to its identifier in the expression where the constant identifier intervenes.

This example also illustrates that certain guidelines can have simultaneous effects on the understanding and on the expression. Indeed, we have pointed out that the use of constant values in the program’s bodies is negative for its readiness, while the use of an identifier of a constant gives this constant a semantic and thus facilitates its understanding.

10.4 FAULT REMOVAL

After having avoided the introduction of faults thanks to valid methods whose essential aim is mastering the design, we are going to examine the obtained *system* to detect the presence of residual faults.

Some verification methods necessitate the specifications, some others methods are led uniquely from the designed system (without referring to its specifications). All these methods do not have the same demonstration strength; they are spread out from a ‘partial functional simulation’ to a ‘complete formal proof’. They apply to the system, which has not yet been fully designed or implemented by an implementation technology (hardware and/or software), or sometimes even to the final product.

We are going to successively examine the verification techniques which make use of the specifications (sub-section 10.4.1), and then the techniques without specifications (sub-section 10.4.2). Each time, we will suppose that the initial model is that of the specifications and the final model that of the system obtained by the design process. However, it is clear that our reasoning applies to any sub-stage of the intermediate transformation, between level I and the following level $I+1$. We provide a panorama of the principal verification approaches without entering in the detail of use of the techniques introduced. However, to make this chapter clearer, we will detail one of these approaches. Furthermore, section 10.5 develop the *functional test* methods by using some examples, and section 10.6 introduces some formal *proof methods*.

10.4.1 Verification with the Specifications

Three groups of techniques explained in the following sub-sections can be imagined for design verification with a specification model:

- by *reverse transformation* of the design model and comparison with the specification model,
- by *double transformation* of the design and specification models into an intermediate model,
- by *double top-down transformations* of the specification model into two design models.

10.4.1.1 Reverse Transformation

This approach goes back from the design model of the system towards the specification model by a transformation, and then compares the result with the specifications (see *Figure 10.4*): $V = D^{-1}$

The V ascending process has to be different from the descending design process, in order to avoid committing the same fault twice. Furthermore, this ascending process can be very complex as the specification and design models are often of a very different nature. For example, how can a structural electronic model constituted of interconnected transistors be transformed into a finite-state machine model? Nonetheless, situations exist where this transformation can be obtained automatically. This is the case when the system is designed as an assembly of interconnected modules, each module being described by a functional logical model (for example an automaton) and the composition of the modules being made of synchronous communications. Example 10.5 (cf. next sub-section) illustrates this automatic transformation mechanism by automaton composition.

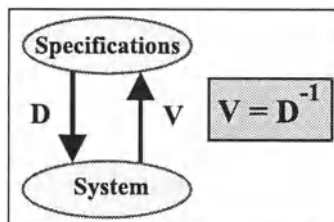


Figure 10.4. Verification by reverse process

On the contrary, in other cases the reverse transformation is made difficult by the use in modules of constructions whose composition is difficult. This is the case of the manipulation of data or constraints (see Example 10.5). The inverse transformation operation is impossible to automate if informal annotations are used in the design model. This situation is very frequent because the models only consider one point of view (or abstraction) of a system. The other aspects have to be added in an informal manner. For example, the behavior of a subprogram, which should be

developed in the next design stage, is simply described by a comment or even just its identifier, such as:

```
Average_Computation(First_Value, Second_Value,
                    Average_Of_The_Two_Values)
```

This informal information is however indispensable and will intervene in the following design stages where their meaning will therefore be formalized as design model. However, as they are by nature informal, these pieces of information cannot be automatically exploited in a reverse transformation.

Finally, even if we can obtain a reverse transformation automatically, we must then confront the obtained model with the original specification model. It is then often necessary to compare two different formulations of one system: the specifications and the result of the reverse transformation. This functional comparison is not always easy. The comparison of two automata can be simple if they are structurally equivalent: we find the same states and arcs. It is slightly more difficult when they are not structurally equivalent. A common standardized and canonical form must be used.

This is a design *proof* if the reverse transformation and the model comparison are formal. We insist again on the fact that, in practice, this analysis is generally difficult to perform.

In the domain of integrated circuits, some methods use *extraction* process. We start from the electronic layout structure and identify the gate networks and other logical modules, as well as their interconnections. Then, we extract the logical combinational and sequential functions to end up with the original logical forms. The results are then compared with the logical specifications (Boolean expressions or logic diagrams).

Example 10.4. Full-adder: logical extraction

Once again, we consider the simple adder already studied in Chapter 5, with three inputs (a , b , c) and two outputs (S : sum, C : carry). Starting with formal specifications in the form of logical expressions ($S = a \oplus b \oplus c$, $C = Maj(a, b, c) = a.b + a.c + b.c$), we have performed the design of this circuit in two stages, as shown in *Figure 10.5*:

- evolution towards the functional level by using two ‘half-adders’,
- evolution towards the logical gate level, ending up with a circuit using several NAND and XOR gates.

This logical design should be followed by an electronic level (by replacing, for example, the gates by MOS transistor structures), then finally by a technological level, leading to a layout, that is to say the floor planning geometric definition of the final integrated circuit.

Now, we will proceed to a reverse operation: analysis of the logical gate circuit, and extraction of the logical functions of the two S and C outputs of

the adder. This extraction can pass through the intermediate stage noted as 'functional' used during the design. It is also possible to directly return to the specification level. We find: $S = a'.b.c + a.b'.c + a.b.c'$ and $R = a.b + a.c + b.c$. These two logical expressions are finally compared with the logical expressions of the specifications. This comparison is rather simple here: the canonical comparison model can be the truth table.

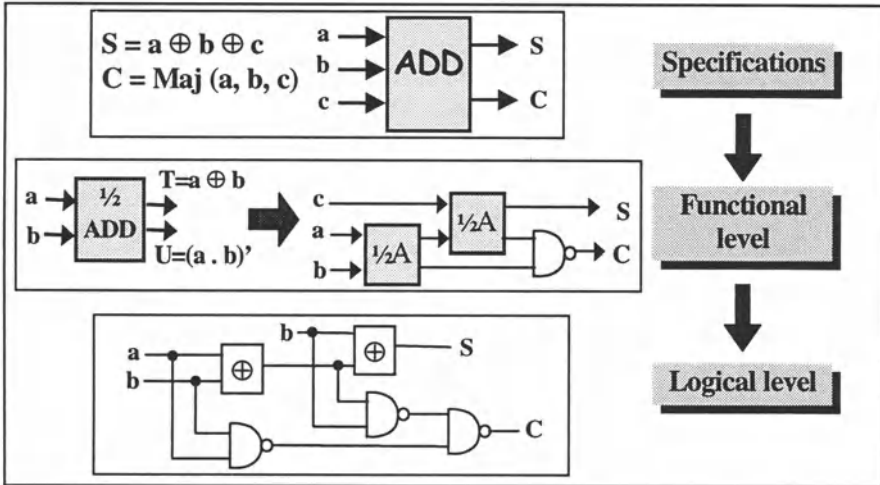


Figure 10.5. Logical design of an adder

Example 10.5. The drinks machine

Figure 10.6 shows a design structure of a variant of the drinks distributor. After a design stage, the structure has three interconnected modules: the *Money_changer*, the *Selection* module and the drinks *Distributor*.

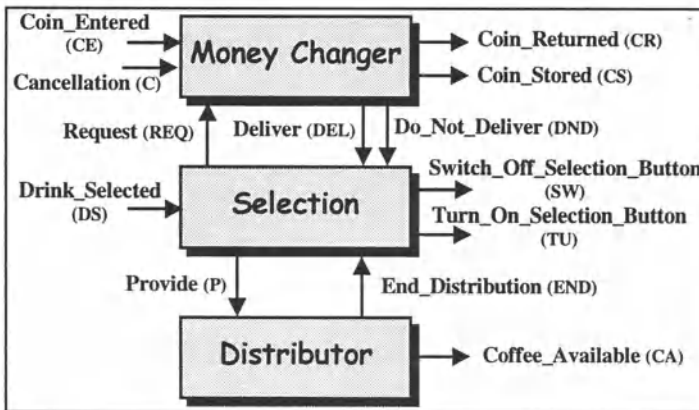


Figure 10.6. General structure

The behavior of each module is described here by a finite state machine (see *Figure 10.7* and *Figure 10.8*). An arc without an original state figures the initial state. The arcs between the modules in *Figure 10.6* indicate the signals exchanged between the modules or with the synchronous external environment. The character ‘?’ defines that the signal is waited to go through the transition, whilst the character ‘!’ states that the signal is sent during the firing of the transition.

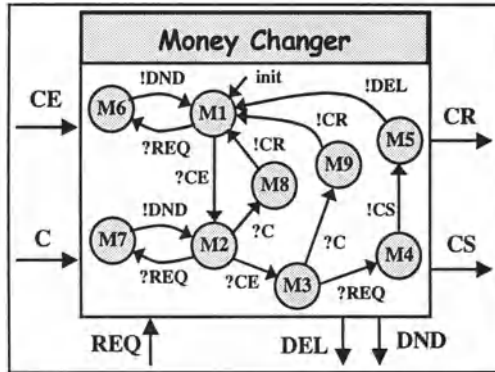


Figure 10.7. FSM of the Money_Changer module

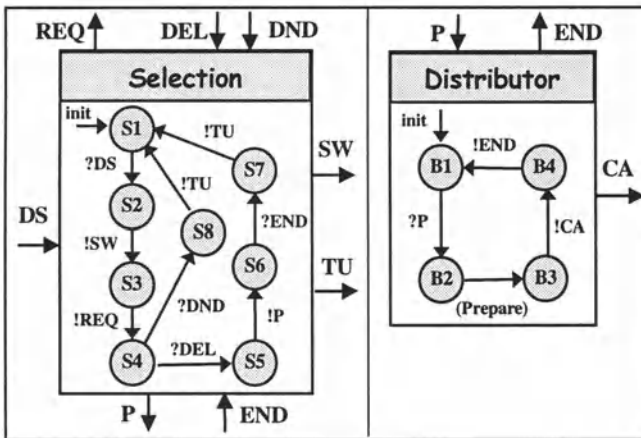


Figure 10.8. FSMs of modules Selection and Distributor

Figure 10.9 provides the product’s complete interface with its environment. The 3 external inputs of the product are: Coin_Entered (noted CE), Cancellation (C), Drink_Selected (DS). The 5 external (or primary) outputs are:

Coins_Returned (CR), Coins_Stored (CS), Switch_Off_Selection_Button (SW), Turn_On_Selection_Button (TU) and Coffee_Available (CA).

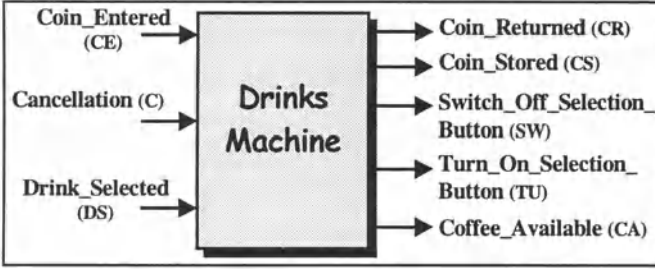


Figure 10.9. Input/output signals of the machine

Finally, we suppose that the functional definition of this distributor has been established during the specification stage. This is provided as a Finite State Machine in Figure 10.10.

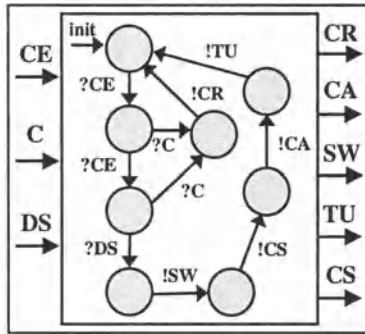


Figure 10.10. Behavioral model of the specification

The real behavior stemming from this design can be obtained by the composition of the automata of the three modules. Several methods exist to carry this composition out. We will use in this example an approach based on simulation. The initial global state of the system is $G1 = (M1, S1, B1)$. Then, we examine the states reached when an input signal is applied. If, for example, the signal Drink_Selected (DS) appears, we go to the global state $G2 = (M1, S2, B1)$, the signal Switch_Off_Selection_Button (SW) is sent at the output, and we then pass to state $G3 = (M1, S3, B1)$. In this state, the Selection module sends the internal signal Request (REQ) to the Money_Changer module and we go to state $G4 = (M6, S4, B1)$. The Money_Changer sends Do_Not_Deliver (DND) which waits for the Selection module. We then go to the state $G5 = (M1, S8, B1)$, and the signal Switch_On_Selection_Button (SW) is activated. Finally, we come back to the initial global stable state: $G1 = (M1, S1, B1)$.

Let us synthesize this functioning extract. From the state noted as $G1$, the signal Drink_Selected (DS) leads to $G2$ where the output Switch_Off_

Selection_Button (*SW*) is sent, then we go to state *G3*. In this state, a sequence of states, internal to the product, takes place which passes from *G4* to *G5*. The signals exchanged between the internal modules are not visible externally, as for all internal transitions. Only the sending of the signal Turn_On_Selection_Button (*TU*) followed by the return to *G1* is perceptible. Considering the specification level, that is an external viewpoint, this internal evolution is equivalent to one single transition from *G3* to *G1*. *Figure 10.11* synthesizes the studied external behavior.

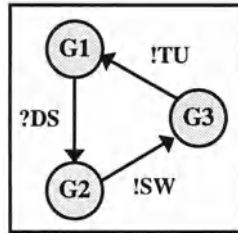


Figure 10.11. Extract of the resulting global behavior

This analysis can be followed on to provide the global behavior by making an abstraction of the internal signals and transitions which are non-perceptible from the outside.

The obtained behavior has therefore got to be compared to the behavioral specification model. The graph in *Figure 10.10* provides this model. It shows that from the initial state, the signal Drink_Selected (*DS*) has no effect: there is no arc labeled *?DS* which leaves from this state. The starting of the automaton's composition which we have just carried out indicates a different behavior: the selection button's light switches off and then on! It is fairly common that the design leads us to specify, or rather choose, behaviors which have not been planned in the specifications and which normally corresponds to functional redundancy. The real problem is to know if the designed system's actual behavior is acceptable to the client.

The operation of reverse transformation (from the designed system towards the product specifications) will be more complex, if, for example, we complete the coffee distributor by the management of coins of different values. Indeed, the 'data part' (leading to data calculation) which completes the 'control part' (pure automaton) renders the analysis process difficult.

10.4.1.2 Double Transformation

The second class of verification consists in transforming, on the one hand the specifications, and on the other hand the system in an intermediate model which is more convenient to perform comparison (see *Figure 10.12*).

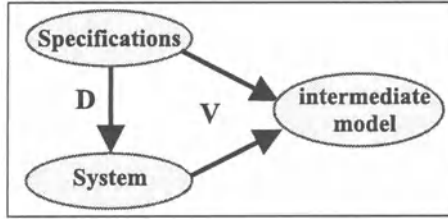


Figure 10.12. Verification with intermediate model

As an example, if a Petri net is used as specification model, and if the design model is a logical gate network, we can decide to take Finite-State Machines (FSM) as intermediate model (Figure 10.13). From the Petri net, we can deduce a FSM called a ‘marking graph’, for example by simulation. Furthermore, we can extract the FSM from the logical model of a gate circuit. Now, we have to compare these two resulting FSMs. We are therefore brought back to the problem raised in the previous paragraph.

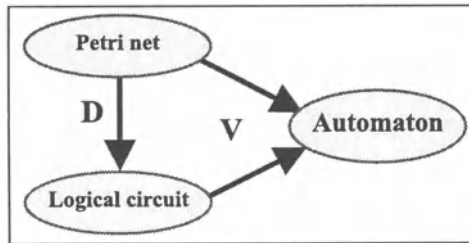


Figure 10.13. Example

10.4.1.3 Top-Down Transformation

The third approach operates using a second top-down transformation V led in parallel with the design phase (see Figure 10.14).

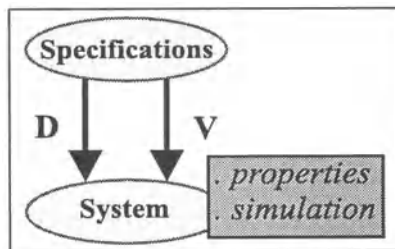


Figure 10.14. Top-down verification

The result of this second transformation is:

- a *simulation sequence*, ordered list of input and output vectors, or
- a set of *functional static or dynamic properties* that the system must satisfy.

These two examples are examined in the two following sub-sections. For both situations, a means must be provided to compare the results of the two transformations.

Simulation sequence

The idea of this method is to deduce a sequence of couples (applied inputs / expected outputs) from the specification model. This *simulation sequence* is also called *test sequence*. To compare the two results, the input sequence is then applied to the design model (the *system*), which in return provides an output sequence compared to the expected outputs. If they are the same, we assume that the system is correct, that is to say that it conforms to its specifications. Of course, the value of this conclusion depends on the quality of the applied sequence. Indeed, if the input sequence only exercises some of the aspects of the system's behavior, the conformity to this behavior only proves the good design of these aspects! This approach supposes that the system model is executable. It is a *dynamic analysis*, as discussed in Chapter 6. We qualify this approach as *functional test* as it uses the specification model, which is a priori *functional*. Often, by extension, the test sequences are defined from *structuro-functional* models, that is to say models which exploit a knowledge of the system organized into interconnected modules. According to the level of structural knowledge about the analyzed system, we speak of:

- *black box test* (no structural knowledge),
- *gray box test* (the system is organized into interconnected modules),
- or *white box test* (we know the whole structure).

We will come back to the *functional test* in section 10.5 by analyzing an example. The *structural test* will be studied in Chapters 12 and 13 for the *verification* of the manufacturing and production stages, as it has been developed to check these stages.

We should note that simulation still constitutes today the most used approach in all domains:

- either on a executable model (executable on a computer) of the designed system's, by applying significant functioning sequences and by comparing the results provided by the computer with those expected (e.g. we simulate an electronic circuit at logic or MOS level),
- or on a physical model (called a mock-up) which is subjected to tests.

Property satisfaction

The functional test seeks to stimulate all the behaviors of the system specifications. The approach known as *property satisfaction* aims at demonstrating that some specific properties deduced from the specifications are satisfied by the designed system.

Consider that a level crossing control system reacts to the detection of trains approaching and leaving by acting on the barrier by actions ‘rise’ and ‘go down’. The difficulty in defining a test sequence is that one or more trains can arrive in the railway section which separates the two sensors (‘approach’ and ‘leave’ sensors). Therefore, it is necessary to test all the possible cases: theoretically, the number of cases is infinite! In fact, the two properties, which we need to verify, are the following:

The barrier is closed when a first train enters in the section.

The barrier stays closed as long as a train is still in the section.

The verification of the two previous properties is critical. The checking of the following one is useful:

The barrier is up is no trains are in the section.

Another example is that of a company’s accountancy management system. The possibilities that such a system offers can be very complex. However, we would like to know if the following assertion is true:

Any order delivered has to be paid before being classed.

A last property example concerns a gas pump management system:

The pump counter cannot be reinitialized as long as the previous client has not paid.

In these three examples, we see that we do not seek to demonstrate the conformity of the system’s behavior with regard to the whole set of its specifications. We wish only to prove that the designed system satisfies certain important and particular characteristics. Therefore, these techniques should not be opposed to the test techniques. Indeed, in section 10.5 and in Chapter 12 we will see the difficulty experienced in demonstrating the exhaustivity of test. The formal demonstration of some critical properties is therefore a supplementary tool. The methods used to demonstrate these properties naturally depends on the specification model and the design model. Some examples of proof methods are given in section 10.6.

10.4.2 Fault Removal without Specifications

Fault removal techniques without specifications aim at checking that the designed system satisfies *generic properties*, that is properties independent

of the specifications (see *Figure 10.15*). In general, we seek to show the existence or the absence of undesired properties such as, for example:

- a functioning deadlock: the system blocks in a state and no longer reacts when input signals occur,
- the system is not living: the property ‘we can go from any state to any other state by an external sequence’ is no longer true,
- the functioning is non-determinist: e.g. an input can provoke different effects from the same internal state.

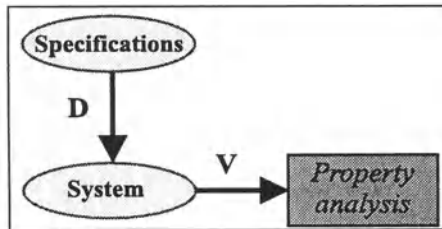


Figure 10.15. Verification without specifications

These properties define potential errors independent from the system’s functionality. The faults at the origin of these bad properties are varied and will probably necessitate a human diagnosis.

A non-living system possesses non-accessible dead parts, which is a clue of a redundant design. For example, one branch of an *if* statement (*then* or *else* parts) can never be run. This situation might or might not be due to a fault. Indeed, redundancy is not always the expression of a fault. For example, redundancy is becoming more and more frequent due to the reuse of hardware or software components which were bought or developed for a previous application. However, the designer should know that certain parts of a reused component cannot be used and therefore justifies the reason for this redundancy. Indeed, dead parts may also be due to a design fault.

The non-determinism of functioning can arise from a design with faults. This is the case with hazard phenomena in sequential systems: the functioning is different according to the temporal values of the component’s reaction. These values depend on the non-functional environment: the time which passes, temperature, etc. This situation arises also for real-time programs whose executive environments can behave in a way which seems hazardous. Indeed, some functioning parameters they use are not perceptible and controllable by the applicative software. For example, the attribution of the processor to a task can be interrupted by the occurrence of an external event. This corresponds to a situation which is defined during the design. This situation can also be due to a preemption mechanism of the real-time

kernel when this software considers that the time allocated to the task is excessive. Hence, this mechanism is not managed by the application whose behavior seems hazardous. This type of property analysis can be carried out on 'state graph' models or mainly on Petri net models if the design implies parallelism and competition.

Example 10.6. Graph properties

Imagine a sequential system's graph with 6 states (*Figure 10.16*). Without knowing the specifications or the product's application, we can analyze some properties of the graph. We notice that it has connectivity problem: if we split the graph into two graphs, $SG1 = \{1, 2, 3\}$ and $SG2 = \{4, 5, 6\}$, it is impossible to pass from $SG2$ to $SG1$. Additionally, state 4 (well state) leads to a blocking, because the system behavior cannot leave this state. Consequently, we should question this system's design as it is not living. This example is considered in Exercise 10.6.

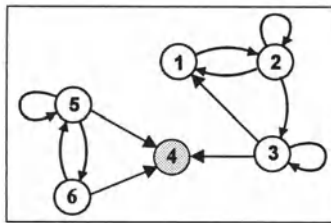


Figure 10.16. Example of graph properties

10.5 FUNCTIONAL TEST

In sub-sections 10.4.1 and 10.4.2 we introduced diverse classes of fault removal means. We also gave an overall view of the principles of these techniques. In this section, we refine the *functional test* technique introduced in sub-section 10.4.1.3. Remember that the functional test consists in applying a sequence of input values to a designed system; the aim of this input sequence is to provoke all the different behaviors described in the specifications. The output values obtained by the system's execution (on a simulation model, a mock-up, or the final product) are therefore compared to the ones predicted, that is to say derived from the specifications.

10.5.1 Input Sequence

One first question is: 'how can we find the input sequence which provokes all the behavioral possibilities?'

When the specification describes sequential behavior, for example using a finite state machine, the test sequence has to pass by all the states and all the arcs of the state graph.

Example 10.7. Coffee machine: input sequence

Consider the coffee distributor designed in sub-section 10.4.1.1 (Example 10.5) whose behavior specification was described in *Figure 10.10*.

An example of a functional test with 11 lines is given in *Table 10.1*. This sequence corresponds to two aborted cycles (by request to cancel) followed by a complete cycle of coffee distribution.

Num.	Input	Output
1	Coin_Entered	-
2	Cancel	Coins_Returned
3	Coin_Entered	-
4	Coin_Entered	-
5	Cancel	Coins_Returned
6	Coin_Entered	-
7	Coin_Entered	-
8	Drink_Selected	Switch_Off_Button
9	-	Coins_Stored
10	-	Coffee_Available
11	-	Button_Lights

Table 10.1. A functional test sequence of the distributor

We now suppose that the distributor accepts different types of coins: 5c, 10c, 25c, and 50c. This modification can be taken into account by associating a parameter *Value_Coin* with the input *Coin_Entered*. The output *Coins_Returned* now receives the parameter *Value_Coin*. To give the change, this output is activated as many times as necessary.

Without wanting to propose a complete new specification, we provide an extract in *Figure 10.17* which defines the actions *!Switch_Off_Button*, then *!Coins_Stored*, then *!Coffee-Available*, etc. which should be carried out only when the amount provided is superior or equal to the value of a coffee.

This extract is preceded by a part which accumulates the value of the coins entered in the machine (*Figure 10.18*). The functional test which integrates this new functionality becomes more complex. It is theoretically imaginable to try all the possible combinations of coins to obtain the exact price of the coffee (*exhaustive test*). In addition, here the user can put more money in than necessary before selecting the drink. Then, a larger number of

combinations can follow, even if the total number of coins acceptable by the distributor is limited.

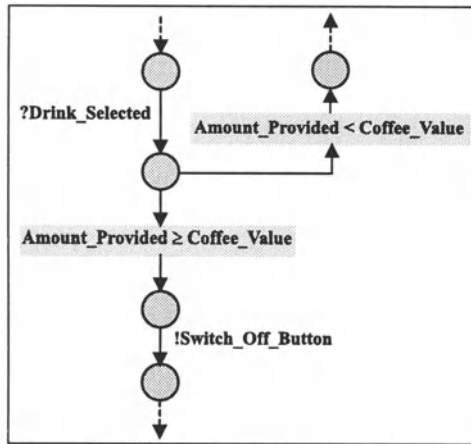


Figure 10.17. Functioning extract of the distributor

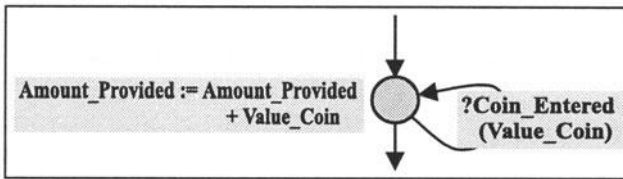


Figure 10.18. Money accumulation

In fact, the specification only considers two cases:

- $\text{Amount_Provided} < \text{Value_Coffee}$, and
- $\text{Amount_Provided} \geq \text{Value_Coffee}$.

If the coffee costs 75c, we can imagine a sequence which consists in putting in 25c, selecting the drink, then waiting for the coin to be returned, then putting in 2 coins of 50c, selecting the drink and waiting for the return of 25c and a coffee. This sequence seems to be sufficient. Indeed, it provokes all the possible specified behavior. However, a question is raised concerning the real demonstration of this property. If the system's designer made a fault when implementing a mechanism that only delivers coffee if the amount provided is *strictly* superior to the amount for the coffee, this sequence cannot show this fault!

When input data intervenes in the specifications, the technique which consists in identifying the value domains which provoke different behaviors, and choosing one value for each of these domains as a test input, has limits

according to the detection of faults in designed systems. This is, however, a very popular method of test sequence.

To improve the test efficiency, engineers often add tests ‘to the limits’ of the value domains. Thus, in our example, the value of `Amount_Provided` defines two domains represented in *Figure 10.19*.

Therefore, we will test the designed system with a value for each domain (for example 50c and 1\$), 0\$ as the inferior limit of domain 1, and 75c as the limit between the two domains. The upper limit of the second domain corresponds to the saturation of the box which stores the coins.

This sequence’s improvement does not, however, cover all possible design faults. Indeed, if we suppose that the 1\$ coins have been added as acceptable by the distributor and that the designer has added a special logic treatment for these coins, a test based on the `Amount_Provided` is not sufficient to reveal the faults having eventually affected this treatment. The functioning of the distributor can be different, according to whether we insert 2 coins of 50c or one coin of 1\$.

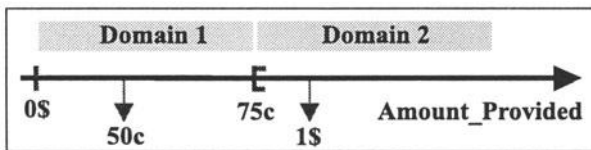


Figure 10.19. Value domains

10.5.2 Output Sequence

The output sequence associated with the input sequence is normally deduced from the specification. However, in certain cases, the outputs are not given by the specifications. This absence of information can arise if the designed system was destined to precisely provide the unknown results. For example, if a physician or a biologist cannot undertake a real experimentation in order to know the effects of the studied phenomenon, he/she would ask for a software simulation of this phenomenon. The expected outputs (produced by simulation) are not known a priori. Hence, the functional test that we have considered cannot be applied so easily. We could therefore attempt to estimate the outputs produced by associating them with intervals, for example. This approach belongs to the *likelihood* group of techniques discussed in Chapter 8 to illustrate functional redundancy and *static* and *dynamic functional domains*. These techniques are close to the verification without specifications discussed in sub-section 10.4.2. For our example, the lack of information concerning the specifications is partial: it is

relative to the outputs of the simulated phenomenon. Likelihood techniques express *properties* on the expected values.

Moreover, we can observe that a test sequence is not only a juxtaposition of an input sequence and an output sequence. It is a sequence of couples (inputs/outputs). To illustrate this, refer to the following example of the coffee distributor.

Example 10.8. Coffee machine: input/output sequence

The possible values of the couples (Coffee_Available, Coins_Returned) make 4 configurations appear (expression of the static domain):

	DS1	DS2	DS3	DS4
Coffee_Available	NO	NO	YES	YES
Coins_Returned	NO	YES	NO	YES

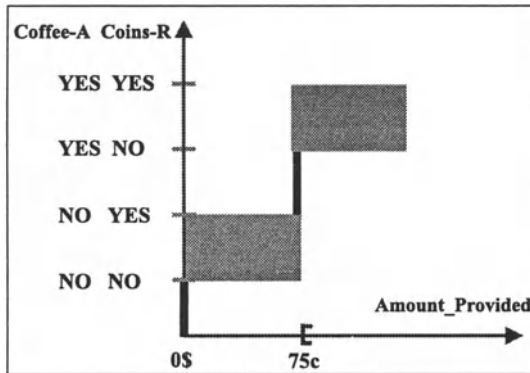


Figure 10.20. I/O domain couples

The behavior defines relationships between the input (Figure 10.19) and output domains (marked above); they are detailed in Figure 10.20. This analysis shows that the test sequence has to cover 4 domains which concern the amount provided before the coffee selection:

- 0\$ no coffee and no coin returned,
-]0, 75c[no coffee and the coins are given back,
- 75c coffee served and no coins given back,
- 75c coffee served and change given back.

Therefore, we see the necessity in testing the cases of 0\$ and 75c, which does not appear explicitly when the input and output domains are studied separately.

Considering the value of the returned amount refines this analysis. This is represented by *Figure 10.21*. When 50c is inserted, this amount must be introduced, and 25c has to be given back. In practice, we do not always consider one single input value per input/output domain. *Figure 10.21* defines the associated input/output values. It is clear that this graphical representation is only possible here because of the simplicity of the relationships between the input and output values.

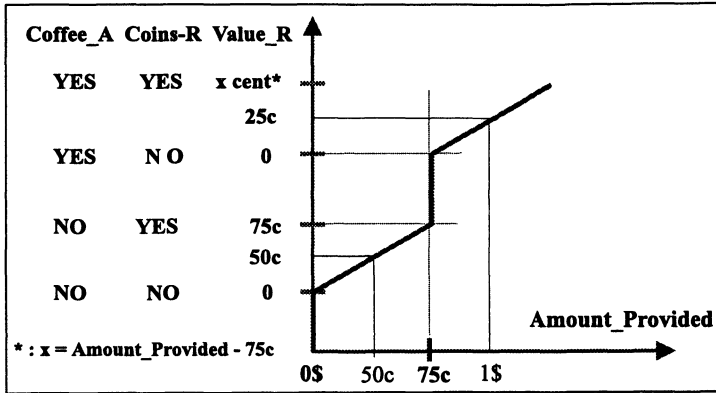


Figure 10.21. Domains and relations

10.5.3 Functional Diagnosis

A functional test sequence allows the presence of faults to be revealed by the failure of the designed system. This failure is detected by a comparison between the expected output values and the values produced at the system’s execution. However, this does not give any information on the fault at the origin of the failure. The *diagnosis* attempts to answer this question:

where is the fault which affects the system?

In this section we are going to give some pieces of information regarding *functional diagnosis* testing, that is to say techniques which try to localize the fault at the functional level. The diagnosis test will be reconsidered in Chapters 12 and 13 concerning its structural aspects.

A first method consists in returning to the causes of the erroneous outputs to discover the infected function. In practice, this backward analysis turns out to be very complex, due to the sequential character of the majority of systems (such as the illustrative coffee distributor used here). In such case, failure stems from a sequential process which activates the fault, then propagates it towards an output (a mechanism already analyzed in the first part). It is therefore necessary to go back in *time* to analyze the system’s successive states without knowing until which previous state to return to!

This approach is also difficult to apply, due to the fact that the system is generally designed from sub-systems (referred to here as modules) interacting. The failure can therefore be due to one of the module's erroneous behavior, but also to the interaction between several modules!

Take the example of the coffee distributor. Its erroneous behavior can arise from a design fault in the interactions between the three modules 'Money_Changer', 'Selection' and 'Distributor'. In practice, two difficulties combine, making the diagnostic extremely difficult:

- go back to the system's history,
- and identify the interaction problems between modules.

A technique which reduces this complexity consists in *instrumenting* the system by adding redundant actions which facilitate the observability of the system's internal evolution and communication between modules. These techniques participate in what we have named as 'easily testable systems'. These techniques will be developed in Chapter 14 for the production and maintenance test.

At the design level of one module, some techniques are proposed:

- the use of *pre-conditions* defining what is expected at the module input,
- the use of *post-conditions* defining what is expected at the output,
- in a more general way, the use of *assertions* on input/output relations.

To illustrate this approach, we consider the example of a subprogram which returns the 'minimum' and 'maximum' values of a list of given values. We can affirm that:

minimum \leq maximum.

This concerns a *post-condition*. It does not give the means to calculate the values produced (which is a design problem), but it gives constraints on its behavior (here on the value of its outputs) when this subprogram intervenes in a program's design. If, during this program's execution, the 'minimum' and 'maximum' do not satisfy the post-condition, this proves the presence of an *error* in the subprogram. In addition, this example shows that the detection is not carried out on the complete system's external outputs but on evaluation of the design modules.

However, it should not be concluded that the subprogram in question is faulty. The fault can concern a bad utilization of this module. For example, we provided an empty list to the subprogram. In conclusion, even if it does not localize automatically the fault which is at the origin of the failure, this technique facilitates, nonetheless, the diagnostic by bringing useful information on the localization of the internal error which has appeared.

10.5.4 Analysis of an Arithmetic Unit

Example 10.9. Arithmetic unit

We consider an addition/subtraction unit in a ‘floating’ normalized decimal scientific system (mantissa in the interval $[0, 1[$). It has been structured by the design into 5 modules shown in *Figure 10.22*. This ‘combinational type’ example completes our illustration of a coffee distributor which is essentially sequential.

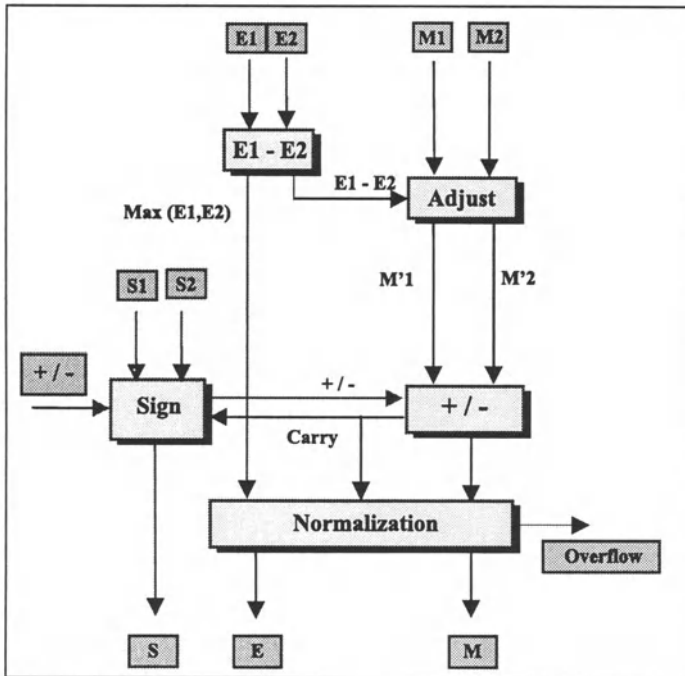


Figure 10.22. Example of a floating arithmetic circuit

The algorithm implemented by this structure proceeds in three phases.

1. First of all, the two numbers must have the same exponent. For this:
 - we compare the exponents (module ‘E1 - E2’),
 - then we adjust the smallest number on the largest number by $|E1 - E2|$ shifts to the right of its mantissa.
2. Then, we carry out the algebraic operation (+ or -) on the two mantissas,
3. Then, we normalize the result and detect a possible overflow.

The module ‘sign’ pilots the operation to carry out on the mantissas and elaborates the sign of the result.

Such a design can be partially verified by a *functional testing*. This is done by making a calculation on a supposedly executable system (mock-up or simulation) and by comparing the obtained result with theoretical value. Furthermore, no knowledge about the circuit's internal organization is exploited by this test. For example:

$$0.1 \cdot 10^{+123} - 0.354 \cdot 10^{+125} = -0.353 \cdot 10^{+125}$$

We ignore a priori the faults detected (or covered) by this elementary functional test but they are probably reduced. In addition, a final non-correct result certainly indicates the presence of an error, but it does not permit the *diagnosis* of a fault.

An exploitation of the structure into modules of the designed system can enrich the verification and permits a more precise diagnosis in case of error. Therefore, this type of simulation is of the 'gray box type' as we observe certain intermediate values. Thus, the simulation of the previous operation will give the output values of each of these modules as well as a final result:

- $E1 - E2 = -2$, therefore the $M1$ mantissa has to be adjusted by two shifts to the right, $M'1 = 0.001$, $\text{Max}(E1, E2) = E2 = 125$,
- the operation control is a subtraction whose result is negative (there is a carry) -0.353 , therefore the final signal has to be corrected,
- finally, there is no need to normalize the obtained result.

Such a simulation is rich in information for the localization of the module or the link between the modules affected by a fault. A study of this example is proposed in Exercise 10.7.

10.6 FORMAL PROOF METHODS

Functional testing aims at showing the system correctness by exercising its behaviors and examining their effects on the outputs. It is the most popular method. In this section, we introduce less popular techniques, based on *formal proof* of properties, which provide additional information in order to improve the confidence in the system correctness.

10.6.1 Inductive Approach and Symbolic Execution

10.6.1.1 Inductive Approach

The formal proof by *inductive approach* aims at demonstrating a conclusion on the system behavior, taking some hypotheses into account. For instance, these hypotheses specify rules on the inputs, and the conclusion

concerns the expected outputs. Hypotheses and conclusions define a *property* on the system. This method is well known in mathematics to demonstrate a theorem. Assuming that an hypothesis is true, we must show that the proposed conclusion is satisfied when the system is executed. For instance, when a train approach is detected (hypothesis), the barrier must go down (conclusion).

The formal demonstration of some properties is very useful, because the client of a product often requires that critical properties be guaranteed. As shown in the previous section, functional testing provokes numerous activations of the system, without giving a formal proof of its correctness (excepted in small system for which all cases can be tested). So, the demonstration of a small number of properties is a useful complementary work. If we note H the hypothesis and C the conclusion, we have to demonstrate that $H \implies C$ when the system is executed.

Example 10.10. Sum of the N first integers

Consider the following program which computes the sum S of the N first integer:

```
S := 0;
I := 0;
while (I<N) loop
  I := I + 1;
  S := S + I;
end loop;
```

The hypothesis is $N \geq 0$ and the conclusion is $S = N * (N + 1)$. This deduction proves the correctness of the program. Indeed, we have:

$$\begin{array}{rcl}
 S & = & 1 + 2 + \dots + (N-1) + N \\
 S & = & N + (N-1) + \dots + 2 + 1, \text{ so} \\
 \hline
 2*S & = & (N+1) + (N+1) + \dots + (N+1) + (N+1), \text{ that is} \\
 S & = & N*(N+1)/2
 \end{array}$$

To make this demonstration easier, we introduce assertions A_i in the system structure. A_1 is the hypothesis and the last assertion is the conclusion. For instance, consider the following annotations of the previous program:

```
-- Assertion A1: N ≥ 0
S := 0;
I := 0;
-- Assertion A2: I ≤ N and S = 0
while (I<N) loop
-- Assertion A3: I < N and S = I*(I+1)/2
  I := I + 1;
  S := S + I;
end loop;
```


-- Assertion A4: $I = N$ and $S = N*(N+1)/2$

To demonstrate $A1 \implies A4$, when the program is executed, we will make partial demonstrations (like lemmas), considering all the possible functioning cases. Taking the program control flow into account, we must show that:

1. $A1 \implies A2$ after the execution of 'S := 0;' and 'I := 0;';
2. $A2 \implies A3$ when $I < N$,
3. $A3 \implies A3$ when $I < N$ after the execution of 'I := I+1;' and 'S := S+I;', and finally
4. $A2 \implies A4$ if $I \geq N$ after A2.

Demonstration:

1. Is evident.
2. Is evident, as $I = S = 0$.
3. Let Ib and Sb the values of I and S before the execution of the loop statements ($I := I+1$; and $S := S+I$;). The hypothesis is:

$$Ib < N \text{ and } Sb = Ib*(Ib + 1)/2.$$

Due to the execution, $I = Ib+1$ (relation 1) and $S = Sb+I$ (relation 2). Due to the loop condition, $I < N$ (first part of the conclusion). We must now demonstrate $S = I*(I+1)$. The hypothesis is $Sb = Ib*(Ib+1)/2$ (relation 3). Due to the relation 1, $Ib = I-1$, so:

$$Ib*(Ib+1)/2 = (I-1)*(I-1+1)/2 = (I-1)*I/2.$$

Moreover, $Sb = S-I$ (relation 2). So, relation 3 implies $S-I = (I-1)*I/2$, that is: $S = (I-1)*I/2 + I = (I-1)*I/2 + 2*I/2 = (I-1+2)*I/2 = (I+1)*I/2$.

So, $S = I*(I+1)/2$ which is the second member of A3.

4. A3 is expressed by $Ib < N$ (relation 1) and $Sb = Ib*(Ib+1)/2$ (relation 2). Due to the execution, $I = Ib+1$ (relation 3) and $S = Sb+I$ (relation 4). We assume that $I \geq N$ (relation 5). Due to the relations 1 and 3, $I-1 < N$, that is $I < N+1$. This conclusion and the relation 5 imply that $I = N$ which concludes the first part of the demonstration. The second part ($S = N*(N+1)/2$) is then demonstrated as previously (item 3), knowing that $I = N$.
5. $A2 = ((I \leq N) \text{ and } (S = 0))$. As the Boolean condition of the while statement is false, then $(I \geq N)$. We conclude that $I = N$, which is the first part of A4. As $I = 0$, then $N = 0$, so the second equality is true, as $S = 0$.

Thanks to these partial implications, we deduce that $A1 \implies A4$, that is, the conclusion is implied by the hypothesis.

The demonstrative power of this technique is very important. However, it has two main drawbacks:

- the engineer must express the intermediate properties,
- he/she has to handle the deductions $A_i \implies A_{i+1}$.

10.6.1.2 Symbolic Execution

Some tools exist to process these deductions automatically. For instance, *Praxis* provides such a tool based on a subset of the Ada language features.

The inductive process can be treated by a *symbolic execution*. The values of the variables are not propagated through the system structure, as they are unknown. Instead, symbolic variables are propagated. Thus, symbolic expressions are deduced which can be reduced.

Example 10.11. Small extract

Consider for example, the following program extract:

```

b := a;
c := a + b;
if (a = 0) then  d := 2*c;
                else  d := -c;
end if;

```

Let A be the symbolic value of a . After the execution of the program, we obtain the value of d :

1. $d = 2*c$ and $a = 0$, or,
2. $d = -c$ and $a \neq 0$.

To obtain the first condition, we process $b := a$; and $c := a+b$;

So, $b = A$, $c = A+A$ and thus $d = 2*(A+A)$ and $A = 0$.

Reducing the expressions, we obtain $((d = 0) \text{ and } (A = 0))$ (relation 1).

In the same way, the second branch of the *if* statement provides:

$((d = -2*A) \text{ and } (A \neq 0))$ (relation 2).

During the second step, we must demonstrate that:

(hypothesis on A) and $((\text{relation 1}) \text{ or } (\text{relation 2})) \implies (\text{conclusion})$.

For instance, the style of the variable a defines constraints on its values.

The conclusions can be constraints expressed by the values of a and d .

10.6.2 Deductive Approach and FTM

10.6.2.1 Deductive Approach

As previously, an hypothesis, intermediate assertions A_i , and a conclusion are defined. However, whereas the inductive approach aims at

demonstrating that $A_i \implies A_{i+1}$ after the execution of a code fragment F , the *deductive approach* operates backwards.

- from the conclusion A_{i+1} , using a backward execution of F , we deduce the condition C_i which must be true before F is executed.
- then, we demonstrate that $A_i \implies C_i$.

The main problem with this method concerns the backward processing of the condition A_{i+1} . Studies established for some design languages have shown the way C_i is obtained when each feature of the language is executed. We will illustrate this process examining some features of the Ada language.

Example 10.12. Sequence

The feature expressing the sequence (noted ‘;’) does not pose any problems: $C_i = A_{i+1}$. Consider the following program extract:

```
-- A1: x > 7 and y < -2
z := x - y;
-- A2 : z > 0
```

From A_2 and the assignment statement, we deduce $C_1 = x > y$. Then, we deduce $A_1 \implies C_1$.

Example 10.13. Conditional statements

Conditional statements needs the study of two branches. Let us consider:

```
-- A1: (abs(x) > z + 4) and (z > 1) and (x > 0)
if (x < 4) then y := 2 - x;
                else y := x;
end if;
-- A2: y > 1
```

If $y > 1$ and we executed $y := 2 - x$; , therefore $2 - x > 1$, that is $1 > x$. This branch was executed if $x < 4$. So, $x < 4$ and $x < 1$, that is $x < 1$ (relation 1).

If $y > 1$ and we executed $y := x$; , therefore, $x > 1$. This branch was executed if $x \geq 4$. So, $x \geq 4$ and $x > 1$, that is $x \geq 4$ (relation 2).

Now, we must show that $A_1 \implies$ (relation 1) or (relation 2). This is true as $z > 1 \implies \text{abs}(x) > z + 4 > 1 + 4 = 5$. So, $x > 5$, as $x > 0$.

10.6.2.2 Fault Tree Method

The *Fault Tree Method* (FTM) introduced in Chapter 7 is based on a deductive approach; the results of the reasoning process are presented as a tree of Boolean expressions using the operators ‘and’, and ‘or’, and ‘not’. The root expression is the conclusion to be demonstrated. The intermediate nodes are Boolean conditions obtained by analyzing the system structure. The leaves are assumptions (such as preconditions) whose values are known.

Figure 10.23 represents such a tree with a conclusion, two intermediate nodes and five assumptions.

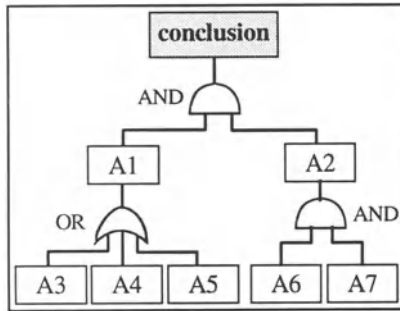


Figure 10.23. Fault tree example

The fault tree method and the associated representations are used here not to specify the causes of a failure but the reasons of an expected conclusion. Let us consider for example, the following procedure specification:

procedure Example (Data: in Type_One; Result: out Type_Two).

Type_Two defines a Boolean condition on the acceptable set of values. For instance, Result is in the range [Vmin, Vmax]. By a deductive approach, we analyze the procedure body using the conclusion on the result. We obtain a tree such as the one in Figure 10.23. Then, we must deduce from the constraints defined by Type_One, constraints on A3, A4, A5, A6 and A7, and finally, demonstrate that the conclusion is always true as A1 and A2 = true.

10.7 EXERCISES

Exercise 10.1. Verification of the adder

Consider the three-bit adder discussed in Example 10.4), and a design fault which consists in replacing in each half-adder the NAND gate with a NOR gate (see Figure 10.5). Study the failures induced by this fault according to each of the three previous verification approaches:

1. by inverse transformation and comparison (functional extraction from the logical gate structure),
2. by double transformation with an intermediate model,
3. finally, by double descending transformation.

Exercise 10.2. Programming style (C language)

We consider the following function written in C language:

```

Min_Max (int table [5], int B)
{
  int i;
  int j = table [0];
  for (i = 1 ; i<5 ; i+1)
  {
    % If B is true, we are finding the min%
    if (B)
      { if (table [i] < j) j = table [i] }
    else { if (table [i] > j) j = table [i]}
  }
  return j ;
}

```

Analyze this function by pointing out the style elements which risk leading to understanding and expression faults.

Exercise 10.3. FSM synthesis

Build the graph in Figure 10.11 (Example 10.5) which describes an extract of the coffee machine's real behavior.

Exercise 10.4. Functional test sequence

Refer to the coffee distributor study. The expression of our client's needs can be summarized to 'make money by providing coffee'. After an initial study, we propose the following informal specification to him:

"To obtain a coffee, the user has to introduce at least 1\$, then validate by pressing on the 'Coffee' button. The change is then given and the coffee served. If the user has not provided enough money before validating, the money is returned and the coffee is not served. The money entered is also returned if we press on the 'Cancellation' button before 'Coffee'. Furthermore, the distributor contains a certain number of coffee doses. The money introduced has to be returned if there are no more doses in reserve."

Using this informal specification, deduce a functional test sequence of the system to realize.

This sequence firstly serves the designer and the client for the verification of the designed system. By defining the input/output relationships, the test sequence also provides the distributor's utilization scenarios. From this point of view, the definition of the test sequence after the specification of the product and its presentation to the client constitute a means of dialogue which permits the verification and the good understanding of needs.

Exercise 10.5. Property research

Using the previous exercise's statement, deduce a fundamental property which the system's behavior must satisfy. Verify that this property is true.

Exercise 10.6. Properties of functional graphs

We return to the graph in Figure 10.16 (Example 10.6). Study its properties:

1. when we delete state 4,
2. when we add an arc which goes from state 5 to state 1.

Exercise 10.7. Verification of a floating point unit

Here, we are interested by the verification of the circuit presented in Figure 10.22 (sub-section 10.5.4) according to a simulation approach. Envisage several scenarios and study their pertinence (capacity to reveal faults).

Exercise 10.8. Inductive formal proof

Let A and B be two positive integers, and Q and R the quotient and the remainder of the division of A by B . Q and R are defined by the property: $A = Q * B + R$. Consider the following program annotated by the assertions $A1, A2, A3$:

```
-- A1: A>0 and B>0
R:= A;
Q:= 0;
while R>=B loop
  -- A2: A = Q * B + R and R>=B
  R:= R - B;
  Q:= Q + 1;
end loop;
-- A3: A = Q * B + R and R<B
```

Show that $A1 \implies A3$ after the program execution. Is this demonstration a proof that the program is correct?

Chapter 11

Prevention of Technological Faults

Stemming from the design stage, the model called *system* has to be transformed into a final product with the aid of technological means which permit its execution in the context of its environment. Two technologies are used in the products studied here, *hardware technology* and *software technology*. In the majority of cases, these two technologies cohabit, their respective weight being dependent on criteria which are often non-functional, such as their speed performance (which favors electronic hardware) or adaptability (more easily obtained by software). We do not discuss these choices, but we focus only on the problems of dependability induced by the use of these two technologies.

This chapter refers to what has been said in Chapter 7 regarding reliability and its evaluation. Examining the implementation stage, it continues Chapters 9 and 10 dealing with fault avoidance in the specification and design phase. Our study focuses on the prevention of *technological faults*. Their removal will be tackled in Chapter 12.

11.1 PARAMETERS OF THE PREVENTION OF TECHNOLOGICAL FAULTS

Technological faults affect a product's functioning after its creation, during the production stage, and during its active life. In Chapter 3, we have already seen that these faults are created by several groups of factors:

- the product's physical characteristics: technology, manufacturing, structure and assembling,

- the environmental characteristics: ageing, temperature, vibrations, shocks, dust, and also frost, aggression by moistures or ‘pirates’, etc. for hardware, and the evolution of execution resources for the software.

First of all, we give some examples from both technologies in order to understand the problems, their differences, and their solutions.

11.1.1 Hardware Technology

As said in Chapter 7, the study of hardware faults is essentially based on statistical data. This leads to representations by functions, curves and estimators of the product’s *reliability* criteria. Time (the cause of wearout) constitutes a fundamental parameter of this criterion: reliability function decreases with time.

Technology has an important impact on component’s reliability: a given CMOS technology has a better intrinsic reliability than others, for example, a better resistance to shorts in the thin oxide structures forming the MOS channels, to cuts in metal lines, to crystalline defects which affect a transistor, to electro-migrations of metal inducing short-circuits, etc. We know that the reduction of the component’s geometric dimensions (shrinking factors) may have negative effects on the reliability of these components by amplifying the *punctual faults* such as local defaults in the semi-conductor’s crystalline structure.

Of course, the real component stems from a technological manufacturing process which also influences its reliability. Badly regulated manufacturing equipment can degrade this reliability. Other important parameters of the final reliability are the product’s structure and the communication signals between modules. Finally, the assembling (connectors, breadboards, etc.) of the modules influences also the final reliability.

Finally, environmental conditions have a large influence on the product’s reliability. In electronics, we note the preponderant influence of the temperature. Are also accounted for in the environment’s parameters that influence the reliability the parameters of the circuit *load*, such as the electrical consumption: its increase reduces reliability. Attempting to improve a final product’s reliability therefore demands the mastering of each of these parameters.

11.1.2 Software Technology

The impact of software technology on product’s dependability is often unknown or underestimated. Indeed, software is not influenced by time or by an increase in the external temperature. However, once again we find ‘technological’ and ‘environmental’ aspects quoted for the hardware

technology, as we are going to show in an example.

Consider a 'real-time' application which uses several periodic tasks T_i (of period P_i) which acquire data from external sensors, treat these data and act on actuators. Suppose that E_i is the duration of T_i operation. If the task management is implemented according to a technique called 'Rate Monotonic Scheduling', we can show formally that the verification of a relationship coupling the P_i and the E_i guarantees that all the data could be treated before their deadline. This condition is:

$$\sum_{i=1}^N E_i / P_i < \ln 2, \text{ where } N \text{ is the number of periodic tasks.}$$

On the contrary, if we choose a different implementation technology, for example an on-line task management, using any real-time kernel, the system can only be verified a posteriori, with many difficulties in establishing exhaustive tests for the real-time applications. The choice of static (of *Rate Monotonic* type) or dynamic scheduling to implement this set of periodic tasks therefore has an influence on the trust that we can place in the software. This example illustrates the impact of the choice of the implementation on the dependability of software applications.

The optimization capacity of the compiler used constitutes an example of the factor associated with the manufacturing or production of the executable code: according to its performance, the generated code will be faster or slower, and thus the real-time application's behavior can change, until becoming non-conform to its specifications (hence a failure appears).

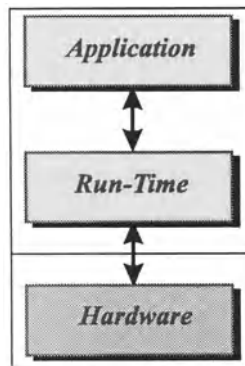


Figure 11.1. Software implementation technology

Finally, concerning the influence of the environment's characteristics, we can again consider a real-time software application. At run-time, this application interacts with a *Run-Time Executive* software (or *executive*

kernel), both software programs being executed on a hardware system (for example a microprocessor). This structure is illustrated in *Figure 11.1*. The behavior and performance of the run-time executive have an impact on the execution time, and therefore on the application's behavior. As a software application can have a very long life cycle, its hardware and software execution environment can vary several times during this life. Thus, during an avionics or automotive system's useful life, the processor and executive software will be changed due to the unavailability of previous versions, improved performance or cost reduction of new versions. The impact of these changes on the product's dependability should be considered during the first realization, in order to avoid failures in the future versions.

11.1.3 Prevention of Technological Faults

Although technological faults occur during operation, their eventuality must however be analyzed from the first stages of the product life cycle. For example, to reduce the appearance of breakdowns makes one choose a high reliability electronic technology, right from the design stage.

Figure 11.2 shows that the protection actions against technological faults start at the specification, and continues during design, production, and finally during operation. Two complementary approaches allow the product's reliability to be improved when faced with these technical problems:

- at the executable product's characteristics level, in order to increase reliability,
- at the environmental characteristics level, in order to preserve the level of the previous reliability.

Actually, product's technological and environmental characteristics are linked: we choose a technology which answers to the constraints of the environment. We separate the two approaches here because in many cases they are not situated at the same level of action or human responsibility.

In this chapter we do not consider the structural approach which, by using *on-line* and *off-line* redundancy, reduces the probability of failures appearing, without reducing the technological faults. This approach of continuity of service acts in masking the faults effect, and does not therefore belong to fault prevention but to *fault tolerance* studied in Chapter 18.

Finally, we note that the prevention of technological faults is complementary to the fault suppression actions examined in the next chapter. These two approaches are often interwoven, in particular by the *quality control* which analyzes certain components (techniques classified in the fault removal approach), allowing an ulterior improvement in the final production (fault prevention).

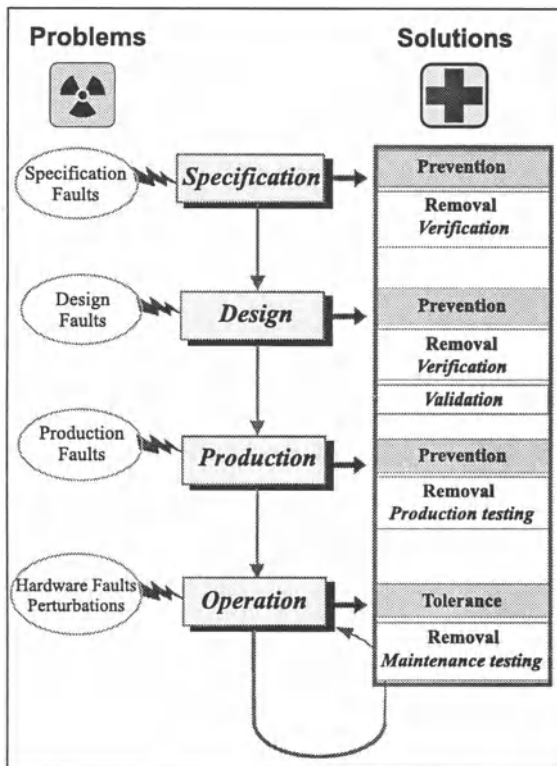


Figure 11.2. Prevention of hardware faults

11.2 ACTION ON THE PRODUCT

11.2.1 Hardware Technology

11.2.1.1 Reliability Law Comparison

Obtaining *high reliability* products starts firstly with research and use of techniques which act on the technological parameters in order to improve the survival of the final product. We know already that the different technological processes which have succeeded during time have brought a considerable improvement to the product's intrinsic reliability. The estimated mean time of correct functioning of today's computers reaches tens of thousandths of hours, whilst it was only about half an hour for the ENIAC in the 40's!

As reliability is a probabilistic parameter according to time, its estimation necessitates measurements on samples representative of the population analyzed (see Chapter 7). From these measurements, we deduce curves

which are used to compare the products associated with one designed system. Thus, in the case shown by *Figure 11.3*, product *P2* has a better reliability than product *P1*, as its survival probability is always superior during time. This improvement results without doubt from the choice of a better technology, and/or a better manufacturing process, and/or a better assembly, and/or due to better utilization conditions.

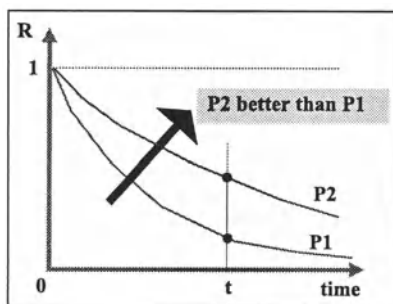


Figure 11.3. Increase of product reliability

The comparison of the reliability of the two products is not always easy, as the reliability curves can cut each other. The example in *Figure 11.4* shows two products whose reliabilities cross at time T : *P2* has a better reliability from $t = 0$ until $t = T$, then it is *P1* which has a better survival probability. Consequently, the choice between these two products depends on the mission's duration: for a mission with duration inferior to T , we would choose the *P2* product. After this, the *P1* product is more interesting.

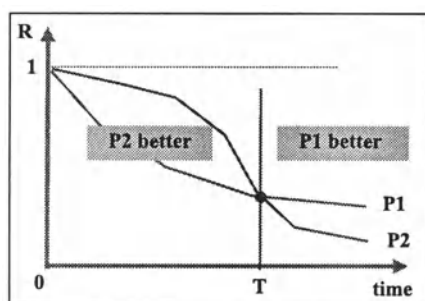


Figure 11.4. Compared reliabilities of two product

11.2.1.2 Reliability Mastering

All industrial domains impose reliability standards on the components used, the severity of which varies from domain to domain. Naturally, the most drastic are the avionics, space and nuclear domains. The price to pay

(technological and human means, as well as the time taken to tune and test) for these reliability requirements is therefore very high! This cost affects the design, manufacturing, and also the on-site implementation phases.

Design Choices

During design, the final product’s reliability is influenced by the choice of appropriate technology. For example, the use of a CMOS technology which reduces the appearance of flaws and which has less susceptibility to parasites. Better reliability can be obtained by *design rules* which impose constraints on the minimal dimensions of electric lines, transistor channels, on the gaps between lines and technological layers, etc.

Thus, an electronic component can have different reliability levels according the technological features involved and the way they are used. Returning to the ENIAC example, the vacuum tubes were not used in their full power, in order to reduce the number of breakdowns. This property was already astutely exploited, many years ago, in the Hammond electronic organs whose reliability at that time was reputed to be excellent.

Production Actions

During manufacturing, reliability is mastered by the use of sophisticated technical means, precise settings of the manufacturing machine’s parameters, quality control methods at different levels of the chain, special soldering and assembly techniques, etc.

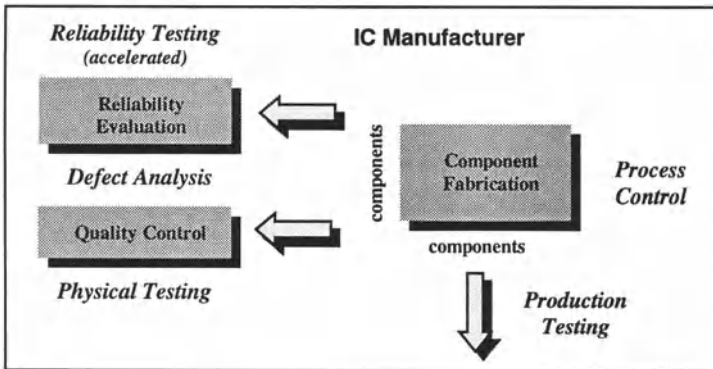


Figure 11.5. Reliability & control evaluation

Several operations are led by the integrated circuit manufacturer in order to evaluate and improve the reliability of the integrated circuits produced. We can identify 4 operation categories illustrated by *Figure 11.5*: reliability evaluation, quality control, process control and production testing. These operations often use test techniques introduced as fault removal means.

However, these techniques are generally applied to product samples, not to remove the defective elements, but to improve the production process in order to prevent the occurrence of faults of future products.

- The **reliability evaluation** means subjects samples representative of the population of product's components to accelerated tests, by an increase in the temperature and/or the power supply.
- The **quality control** techniques consist of a refined technological analysis of components taken in the manufacturing chain, in order to determine the population's quality, and to determine the causes of failures.
- The **process control** techniques apply tests to the manufacturing equipment in order to verify their good functioning.
- The **production testing** means apply physical, chemical, optical and electrical tests to the components during the diverse stages of the production chain, and just before they are released to customers.

Reliability evaluation applies principally **non-destructive** or **destructive accelerated tests** in order to estimate the parameters of the survival law of the population of components. The possible use of destructive test shows clearly that the aim here is not to detect a particular non-reliable product from the manufacturing chain, but to reveal a bad reliability of the ensemble of the products. In the automobile domain, crash tests have the same objective: to verify the good resistance to shocks during the development of the car model, and not of the particular car intended to a client.

Refined physical and chemical complementary tests allow mortality causes to be diagnosed and also to improve the production's reliability. These tests are also used by the quality control operations.

The quality control integrates information coming from diverse sources, in order to verify the product's quality. This information arises from measures carried out on the process itself as well as on the products which stem from it: complete quality control of all incoming materials and monitoring of all wafer and assembly processes.

Therefore, we subject all or part of the product's components to **reliability assurance tests** and **quality assurance tests**. These tests include life tests, mechanical tests, thermal tests, lead fatigue, and solderability tests, as shown by the example in *Figure 11.6*. Some compliance tests may be applied by independent organizations, in order to verify that the circuits satisfy or not given reliability requirements.

The checking and the characterization of integrated circuits are carried out by varied investigation means. For example:

- external inspection with a microscope,
- radiographic inspection,

- control of the rate of leak ratio in sealed packages,
- detection of free particles in the cavities of the packages,
- opening the packages and inspecting the dies,
- control of the mechanical quality of the connection wires and dies.

LIFE TEST	THERMAL TEST
High Temperature Operation	Temperature Cycling
High Temperature Storage	Thermal Shock
High Humidity Storage	Soldering Heat
MECHANICAL TEST	LEAD FATIGUE
Shock	SOLDERABILITY
Constant Acceleration	
Vibration	

Figure 11.6. Quality Assurance Test of ASICs

The *process control* becomes a standard in all the production of integrated circuits. It helps the optimization of reliability through defect reduction: implementation of a variety of particle monitors at each stage of the manufacturing process to prevent, detect, and eliminate the incursion of foreign materials into the process. Wafer scanners use laser beams to detect microscopic particles on the wafer surface. Laser-based particle counters measure the number of particles generated, while local air-borne particle counters placed at strategic locations near processing or handling areas measure particles in the air surrounding the wafers. High sensitive particle test chips are also used to measure process defect densities and validate improvements. When a too high particle level is detected, particle analysis tools are used to identify their cause by in-depth analysis.

Production testing participates in fault removal, and for this reason it will be analyzed in Chapter 12. However, it contributes also to the increase of reliability of manufactured products by signaling the manufacturing process problems. These tests, therefore, can also be used as fault prevention techniques. The aim of such *screening tests* is to remove faulty circuits but also weak circuits able to present infant mortality failures.

11.2.2 Software Technology

In the case of software, the degradation phenomenon does not exist. However, faults related to the programming technology used (e.g. of the language) can arise. In order to prevent these faults, the choice of programming language has to firstly be carried out with precaution.

Secondly, the use of certain of the chosen language's features can be forbidden. Only one restrained version of the language is accepted. Following this, rules impose the way to use the remaining features. Finally, the evaluation means of the written program's dependability are put into place by sample analysis. Regarding the programming process, their conclusions allow improvements to be proposed. These four means of fault prevention associated with the programming language are developed in the following sections. The prevention of faults relative to the run-time environment of these languages will be studied in section 11.3.2.

11.2.2.1 Programming Language Choice

The realization of a system by using a software technology necessitates choosing a programming language. Once the program has been written in this language, the implementation is then led by successive stages (compiling, linking and execution) on which the engineers have few actions. A well-argued analysis of the languages has to be done. The choice of a language has to be justified by dependability criteria. For example, the Ada language is today frequently chosen in numerous domains (aeronautics, space, nuclear, etc.), not for its original features (genericity, protected objects, etc.), but for its intrinsic dependability implied by its features. Without going into an exhaustive study, we simply quote here two examples.

Firstly, Ada offers greatly varied features which permit, for example, to avoid to confuse two conceptually distinct types having the same implementation.

Thus, the statements

```
type Lift_Levels is new integer range 0..9;
type Digits is new integer range 0..9;
```

define two distinct types which do not allow a type expression to be assigned to one variable of the other type, although these two types are implemented in an identical way. In addition, even if these two types inherit values and operations from the general type `integer`, they are still distinct.

We give a second example. Numerous programming faults are due to the use by a component of elements whose access is forbidden. These elements have strictly restricted access rights to other components. The Ada language offers several encapsulation mechanisms, which allow elements to be made private. These elements can, for example, be variables or subprograms. Packages, tasks, and subprograms are examples of features which permit such a protection. For instance, a subprogram `Child` can be included in a subprogram `Father`. Therefore, the `Father` is the only one which can call the `Child`.

```
Procedure Father is
```

```
. . .
```



```

    procedure Child is
        . . .
    begin
        . . .
    end Child;
    . . .
begin
    . . .
    Child; -- subprogram call
    . . .
end Father;

```

11.2.2.2 Restriction of the Features

Once the language has been retained, its features have to be studied in order to exclude those which lead to an increase in the risk of introducing faults. This reduction of the number of language features is therefore a fault prevention action. Here again, we cannot go into detail with an exhaustive study in this book. We will examine only two features by showing why they are dangerous and therefore why their use should be prohibited.

Example 11.1. Shared variables

The programming of multi-tasking applications can often lead to the implementation of communication between tasks by shared variables. The behavior induced by the use of features allowing this programming can be hazardous. We consider two variables $V1$ and $V2$ local to two tasks `Task1` and `Task2`, and a shared variable S . The first task increments the shared variable whilst the second decrements it as shown in the following extract:

```

task Task1 is
    V1: . . .
begin
    . . .
    V1 := S;
    V1 := V1 + 1;
    S := V1;
    . . .
end Task1;
task Task2 is
    V2: . . .
begin
    . . .
    V2 := S;
    V2 := V2 - 1;
    S := V2;
    . . .
end Task2;

```

Figure 11.7 shows three possible sequences of the execution of these two tasks. If the initial value of S equals '3', its final value is '2' in case 1, '4' in case 2, and '3' in case 3. In order to avoid meeting such hazardous functioning, communication by shared variables will be forbidden, even if the chosen language authorizes it.

The problem encountered here arises from the use of the copies $V1$ and $V2$ of S , local to tasks Task1 and Task2 . Exercise 11.3 shows that reading or writing directly in a data structure, without carrying out such copies, leads to other difficulties when the data structures are complex. Exercise 11.4 shows that hazardous phenomena also occur when simple data structure are used without local copies ($S := S + 1$ and $S := S - 1$).

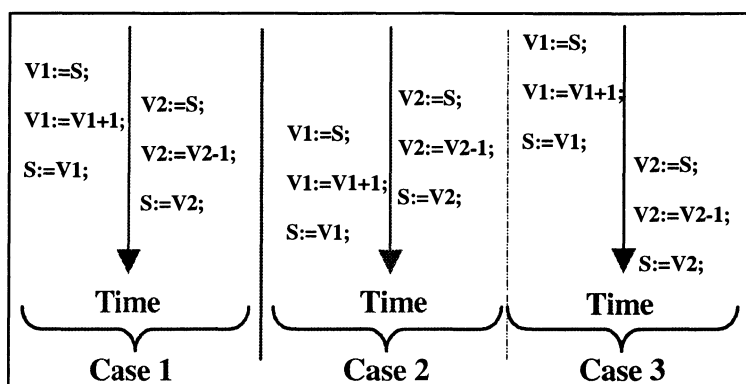


Figure 11.7. Use of shared variables

Example 11.2. Goto statement

The `goto` feature is offered by the majority of programming languages. Its implementation on hardware does not pose any problems. However, its use is imperatively forbidden for dependability reasons. The probability of being the cause of faults is high, as this feature does not allow a clear vision of control flow. Indeed, it creates a rupture in the interwoven structure of the control flow. On the contrary, this structuration is favored by:

- statements such as 'if ... then ... else', and 'for', which offer static overlapping shown in the left part of Figure 11.8, and
- the subprogram whose call mechanism proposes a dynamic overlapping symbolized in the right part of Figure 11.8.

The rupture of this structuration by the `goto` feature is represented in both cases by Figure 11.8. This rupture is in total contradiction with the two abstraction concepts indispensable to the understanding of the programs: 'is composed of' and 'makes use of'.

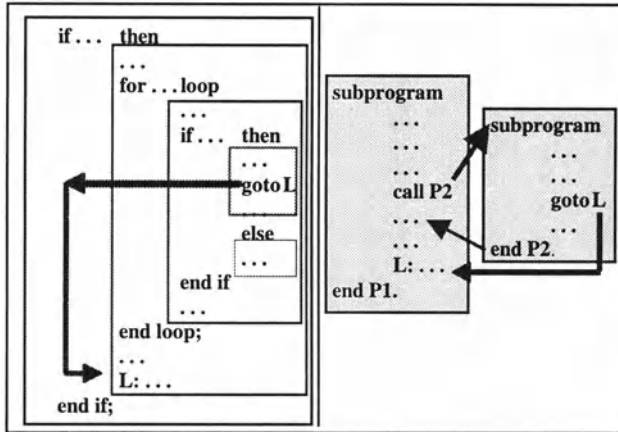


Figure 11.8. Structured control flow

11.2.2.3 Imposing a Programming Style

The way to use the authorized features is often restrained by utilization guidelines. The aim of these guidelines is to limit the risk of technological faults that some uses induce with a high probability.

For example, the use of the statement `return x;` by a function to return a value to the calling program cannot be placed as the last statement of the body of the function. Then, each use of this statement creates a rupture in the control flow; hence, several uses multiply the number of output points of the function, making it more difficult to understand. Consequently, a guideline specifies that the return statement must be the last one in a function body.

We consider the data type `Float` as a second example. Due to rounding errors, two distinct numeric values can be conceptually identical. This is the case of `0.999...999` and `1.0`. If the use of the 'Float' type has not been excluded, the comparison between the two real numbers has to be done by integrating these errors. We will use for example:

```
if abs(V1-V2) < Epsilon    % the two values are equal
instead of
if V1 == V2
```

11.2.2.4 Programming Process Improvement

Programming automation

The production is principally a human activity where source programs are concerned, whilst it is mainly automated for hardware products. Therefore, we seek firstly to prevent faults by automating or by systemizing the programming stage. For example, a tool can carry out the transformation of a design model from a determinist finite automaton into a program.

Example 11.3. A simple automaton

We consider the example of the automaton in *Figure 11.9*. We suppose that ‘Activate’, ‘Stop’, ‘Sleep’, ‘Action’, and ‘Complete’ are 5 services offered by the component. This component accepts to provide these services according to its current state.

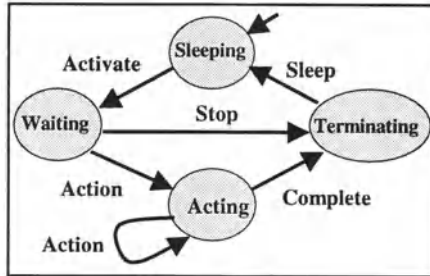


Figure 11.9. Automaton

This behavior is transposable in a systematic way into Ada language by:

```

task body Example is
  type Set_of_States is (Sleeping, Waiting, Acting,
                        Terminating);
  Current_State: Set_of_States := Sleeping;
begin
  loop
    select
      when Current_State = Sleeping
        => accept Activate do
          . . .
          end Activate;
          Current_State:=Waiting;
      or
      when Current_State = Waiting
        => accept Stop do
          . . .
          end Stop;
          Current_State:=Terminating;
      or
      when Current_State = Waiting
        => accept Action do
          . . .
          end Action;
          Current_State:=Acting;
      or
      when Current_State = Acting
        => accept Action do
  
```

```

        . . .
        end Action;
        Current_State:=Acting;
    or
    when Current_State = Acting
        => accept Complete do
        . . .
        end Complete;
        Current_State:=Terminating;
    or
    when Current_State = Terminating
        => accept Sleep do
        . . .
        end Sleep;
        Current_State:=Sleeping;
    end select;
end loop;
end Example;

```

This translation can easily be systematized:

- introduction of a data type (*Set_of_States*) which enumerates the automaton's states,
- introduction of a variable (*Current_State*) which defines the current state, and which is assigned by the initial state,
- introduction of an infinite iteration (*loop*) which allows us to return to the selection statement,
- according to the current state, accept the services associated with the arcs which leave from this state (when *Current_State* = ... => accept ...), carry out the associated action (*do ... end*), then make use of the new current state indicated at the extremity of the arc (*Current_State := ...*).

The program produced is undoubtedly not optimal. For example, the automatic translation leads to the writing of the assignment '*Current_State := Acting;*' after having accepted the service 'Action', whilst the state already had this value (when *Current_State* = *Acting*). The automatization itself is not justified by the search for an optimized program but by the absence of faults in this program.

Programming assessment

As the programming cannot be entirely automated, the programs produced have to be analyzed in order to improve this activity which still remains essentially human.

This analysis is firstly done on program samples during development.

However, we do not possess information about potential faults at the origin of these possible future failures. For this reason, we evaluate if the constraints imposed on the programming style have been respected: non-uses of forbidden features, respect of the utilization rules for the authorized features. If one of the constraints is not respected, we point it out to the engineer so that he or she can modify his/her way of working. We know, indeed, by experience that this non-respect creates a risk of faults. This work can be carried out on program samples as we suppose that the bad work methods are still practiced.

If we possess information regarding the failures of the already operational programs, the analysis of fault causes allows constraints or guides to be added to the programming process. Therefore, we advise to record failures and to maintain a database containing the circumstances and analysis of causes.

11.3 ACTION ON THE ENVIRONMENT

11.3.1 Hardware Technology

For a given technology, we increase the reliability by mastering the parameters of the application's non-functional environment. As these parameters are numerous and varied, it follows that the control means are also very varied. Thus, as we have already pointed out, the temperature is a fundamental parameter of the reliability of electronic components:

the reliability decreases when the temperature rises.

There are laws which permit, for a given technological population, the reliability at a θ_1 temperature to be determined from established databases at θ_2 temperature (*Henry's abacus* quoted in Chapter 7). Different techniques planned during design, production, and possibly during the application's implementation, allow the temperature to be controlled and therefore increase or guarantee a certain level of required reliability:

- a passive or 'natural' control, for example the rotation of an artificial satellite which avoids having the same side exposed to the sun,
- an active or 'artificial' control by the product's air condition: this is the case of microprocessor cooling techniques by a radiator and an air fan, or computer cooling by water used for ECL technology.

In the same manner, perturbations arising from the environment, such as particles or electromagnetic phenomena, have to be analyzed very early on; they lead to various solutions:

- internal by the choice of technologies which are not sensitive to these perturbations,
- or external by the use of shielding techniques.

It should be known that the alpha particles constitute a real problem for DRAM type memory, by provoking the loss of stored data by charge or discharge of the memory points which are of capacitive type. These temporary faults are qualified as *soft faults* (in opposition to permanent faults which are said to be *hard faults*); they can affect industrial applications. In particular, they have been reduced by the use of a passive protection, a layer of resin applied on the integrated circuit which traps these particles. The importance of the efforts made by companies working for aeronautic and space applications in order to counter problems of the EMC (Electro-Magnetic Compatibility) should also be known. International standards that must be satisfied by electronic equipment are being put into place, which allows higher robustness against radiation.

Beyond particular points dealing with temperature or radiation, all operating conditions should be mastered during the creation stages, then controlled during the operation by supervision and maintenance techniques.

11.3.2 Software Technology

The production of an executable program from a program source, and its execution in operational phase are entirely automated. The tools used for this are the following: a compiler, a linker and a Run-Time Executive or Kernel. They are regrouped under the term of Run-Time Environment. Obtaining these tools does not require big efforts from the source program designer. He/she selects them rapidly with the purchase price as the principal criteria, and then the ergonomics of the interface or the quality of the documentation. Dependability issues are generally not considered. In the same way, a Run-Time Environment has to be chosen with care, taking the demands for dependability into account. We are going to present three criteria which should be studied:

- the existence of Run-Time Environment verification means,
- the research and exclusion of language features whose diverse implementations lead to hazardous behavior,
- the definition of constraints on the implementation of the language's features in such a way which warns against the effect of perturbations altering hardware resources.

11.3.2.1 Confidence in the Run-Time Environment

The trust attributed to a chosen Run-Time Environment language has to be justified. Indeed, the presence of a fault in a compiler, a linker, or an executive can induce a failure in the application. This trust can be established by tests in which the inputs are constituted of a set of programs which manipulate the possibilities of the programming language. Due to the multitude of the possible feature combinations offered by the language, the realization of exhaustive tests constitutes an enormous and non-directly productive work for the firm. Not being able to carry out these tests, this firm disposes of two complementary solutions:

- use of a language whose validation procedure for run-time environments already exists and which is accessible (this is the case, for example, with Ada language),
- keep as much as possible, during several projects, an already used environment whose list of usage situations creating erroneous programs is maintained. This information arises from the manufacturer and from the environment's user groups. This has to be added to the knowledge of the engineers who are responsible for the program development.

In this last case, even if the environment present flaws, they are identified and we can have faith in the use of language from which have been forbidden the yet recorded dangerous configurations. We should point out that, contrarily to the first solution (environment validation procedure), it is the compiler's clients who (unfortunately) are frequently the testers. The need for dependability therefore justifies the fact that a company refuses to use the 'last compiler which has just come out'.

11.3.2.2 Hazardous Features

Certain features of languages are not conceptually dangerous. For this reason, they have been preserved after their study which was exposed in section 11.2.2.2. However, these features have to be excluded, due to the variable behavior they induce in the executable program, according to the realization choices of the Run-Time Environment.

Pay attention! The real effect of a feature on an executable program's functioning can vary. This happens not only when passing from one environment provider to another, but also for two environments which come from the same provider. These two environments can be distinguished by different hardware platforms or by different versions on the same platform.

We consider real numbers as a first example. Their mathematical definition is very precise. However, a computer cannot transpose this definition. For example, the series $S_N = 1/1 + 1/2 + 1/3 + \dots + 1/N$ is

mathematically divergent whilst it converges when it is translated in the form of a program. Indeed, when N is big, $1/N$ is assimilated to 0.0. In addition, the value to which the program converges depends on the definition of calculation means offered by the Run-Time Executive.

We consider a second example of a program which uses shared variables. We suppose that no interleaving exists between the used features as described in sub-section 11.2.2.2. Therefore, there are no design faults. However, its implementation on a distributed system can be carried out using several exemplars of the variable. A copy for each machine executing at least one task which makes reference to this variable will indeed increase the performance. However, the integrity of the copied values is not always guaranteed. Only the 'synchronization points' between tasks generally ensure bringing all the diverse copies up to date. It seems that this situation cannot be met in the case of a simple variable (contained in a memory word) and memorized on a single machine. Exercise 11.4 shows that this problem can also exist in this case, even on a mono-processor system.

11.3.2.3 Implementation Constraints

In the case of software technology, restrictions can be made to the way a feature is implemented on a given hardware platform. These constraints apply for example to the code generated by the compilation of a language statement. They aim at obtaining a unique behavior and to support the perturbations of the execution resources. Once again, the demands of dependability will eventually go against the demands for performance.

To illustrate these constraints, consider the example of the classical multiple branching statement (statement *Case* of Ada language, or *Switch* of C language).

Example 11.4

```

case Choice is
  when Choice = 1   => Treatment_1;
  . . .
  when Choice = N   => Treatment_N;
  when others      => Treatment_others;
end;
```

Two techniques are generally used to implement this statement.

The first one, called 'branching by address table', uses a table, which, at every I value of the `Choice` variable, makes the address correspond to where `Treatment_I` code starts. If `Choice` can take more than M values, with $M > N$, the addressing table contains the addresses of the start of `Treatment_Others` for the $M-N$ last cases. This solution produces a high-

performing final executable code, as the addressing is direct according to the I value of the Choice index.

The second solution treats this statement as a set of nested if:

```

if Choice = Choice_1 then Treatment_1;
elseif Choice = Choice_2 then Treatment_2;
. . .
elseif Choice = Choice_N then Treatment_N;
else Treatment_Others;
end if;

```

The execution of the code thus generated is a lot slower. Indeed, before starting the execution of Treatment_I, it should be stated that: Choice \neq Choice_1, then Choice \neq Choice_2, then Choice \neq Choice_3, . . . , and finally Choice \neq Choice_I-1.

However, if the value of the Choice variable does not belong to the value intervals (Choice_1, . . . , Choice_N), nor to $(M - N)$ other possible values (branch Treatment_Others), due to a previous erroneous calculation or due to a fault in the memory containing the value of Choice, therefore:

- with the first solution, the branching will be carried out anywhere (at a memory address outside the table, non-specified),
- with the second solution, the erroneous value of the Choice variable will provoke the execution of Treatment_Others.

In the first case, the program's real behavior will be hazardous, In the second case, it is known a priori. Therefore, the Treatment_Others can be reserved to an error treatment.

11.4 EXERCISES

Exercise 11.1. Component choice

A product can be realized according to two non-redundant and functionally equivalent structures which use components of reliability laws with constant failure rate. The first solution $S1$ has:

- 12 components with a failure rate of 10^{-7} ,
- 1 component with a failure rate of 10^{-6} ,
- and 3 components with a failure rate of 10^{-5} .

The second solution $S2$ has 4 components with a failure rate of 10^{-6} .

Which one is the best choice from the reliability point of view?

Exercise 11.2. Comparison of the reliability of two products

Two products, $P1$ and $P2$, have exponential reliability laws of respective failure rates $\lambda_1 = 10^{-5}$, and $\lambda_2 = 10^{-7}$, at a temperature of 18°C . We suppose that the failure rate is multiplied by 10 for an increase of temperature of 20°C for the product $P1$ and of 10°C for $P2$. Furthermore, the products have a maximum temperature of good functioning of 100°C for $P2$ and of 120°C for $P1$.

For which temperature range does the $P1$ product have a better reliability than the $P2$ product?

Exercise 11.3. Shared FIFO

The asynchronous communication between real-time application's tasks is programmed by using a FIFO list (First In - First Out). This list is implemented by a data structure composed of an array named Buffer of size Buffer_Size with two indexes Write_Index and Read_Index. These indexes respectively represent the index of the first free place in the array, and index of the next character to read. In the zone of the array between Read_Index and Write_Index-1, values are stored as shown in Figure 11.10.

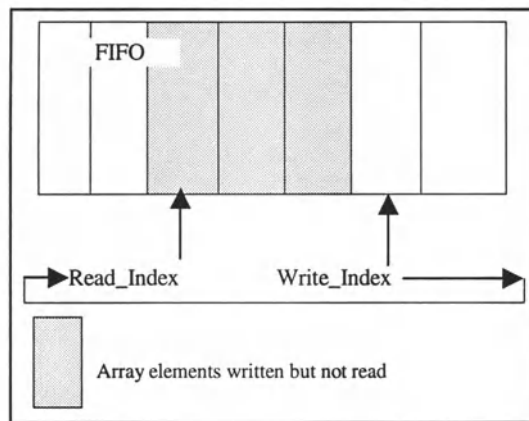


Figure 11.10. FIFO management

We suppose that there is always at least one element in the array and that it is never full. This array is managed like a circular buffer. Thus, the access operations write and Read are programmed by:

```

Procedure Write(X : in Element) is
begin
  Buffer(Write_Index) := X;
  Write_Index := (Write_Index mod Buffer_Size) + 1;
end Write;

```

```

Procedure Read(X : out Element) is
begin
  X := Buffer(Read_Index);
  Read_Index := (Read_Index mod Buffer_Size) + 1;
end Read;

```

Determine the state of the array when:

1. two calls to `Write` happen at the same time (the statements of the body of the procedure `Write` are executed in an interwoven way);
2. a call to `Write` and a call to `Read` happen at the same time (the statements of the bodies of the procedure `Write` and `Read` are executed in an interwoven way).

What would you conclude about this programming of an asynchronous communication between two tasks?

Exercise 11.4. Hazards in shared variable implementation

We consider a multi-task application which uses a shared simple type variable, that is to say memorized in one single memory word. We will also suppose that all the modifications of this value are carried out directly, without local copies in the tasks, contrarily to what was presented in section 11.2.2.2. No design or programming problems exist therefore, and this communication mechanism between tasks seems to be without risk. Its utilization should even be encouraged due to its performance. This exercise aims at showing that faults are introduced by the implementation of this mechanism.

We study the implementation of reading and assignment statements for this variable. We will suppose that the microprocessor offers `decrease` and `increase` statements which act uniquely on the registers. By studying the code generated by the compiler for the extracts of two tasks, show how this brings us to the problem discussed in section 11.2.2.2.

Program:

```

Task 1 :
  . . .
  I++;
  . . .
end Task1.
Task 2 :
  . . .
  I--;
  . . .
end Task2.

```

Chapter 12

Removal of Technological Faults

We continue our exploration of the principal groups of protection methods by now considering *dynamic analysis* techniques, called here *off-line testing* techniques. *Test sequences* are applied to an *executable product* during the production and operation stages. This chapter dealing with the removal of technological faults extends Chapter 11 dedicated to prevention of technological faults. However, the techniques presented here also allow the detection of certain functional faults stemming from previous specification and design stages, faults which occurred despite the protection means used during these stages. We note that these residual faults should have been detected earlier, as their detection in the production, or, even worse, in the operation phase, may question the design and technology choices. This chapter is principally concerned with hardware products. In Chapter 13, we will complete this presentation of on-line testing with the study of several simple structural test methods for hardware and software systems.

Section 12.1 provides a general overview of off-line testing and the relationships between the product and the tester. In section 12.2, we focus our study on the specificity of logical production and maintenance testing. Finally, the problem of generating tests on logical circuits at the gate level is considered in section 12.3.

12.1 OFF-LINE TESTING

Test plays a major role amongst the *fault removal* approaches. In Chapter 10, we have already discussed this point, in regard to the design stage.

12.1.1 Context of Off-Line Testing

Here we focus on the *production* and *operation* stages (Figure 12.1) which imply different test methods known respectively as *production testing*, and *maintenance testing*.

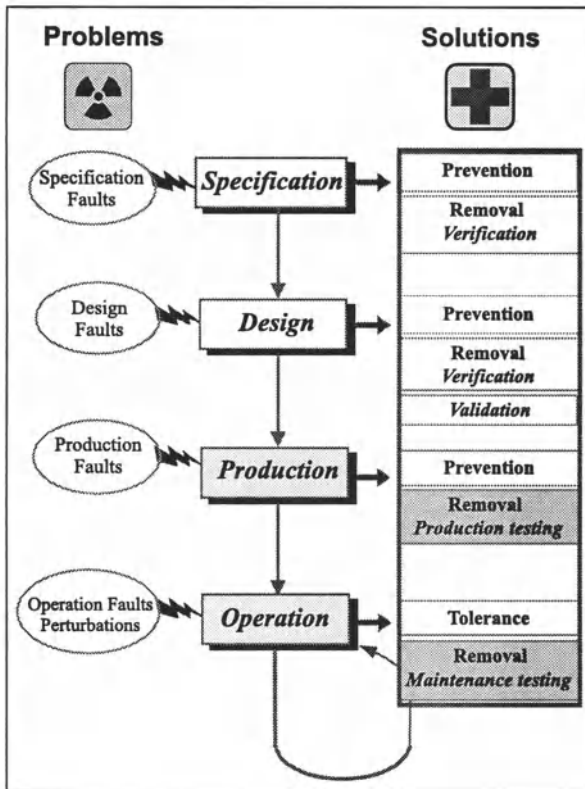


Figure 12.1. Production and maintenance testing

Test is originally an experience led externally on a real system to confirm or invalidate a hypothesis or to distinguish between several hypotheses. This word has been first developed in the biological and psychological domains. We thus apply *stimuli*, that is to say a set of sequences constituted of input vectors, and we observe and interpret the output responses by comparison with the expected values. In our context, we will subject a product to an *experiment* in order to determine if it functions correctly or not, and eventually identify the fault or faults affecting it.

Two kinds of tests exist:

- The *detection test*, which answers the question: does the product function correctly?

- The **diagnosis test**, or **localization test**, or **debugging**, which answers the question:
which faults affect the product?

These two test categories are different. They are traditionally linked according to a scenario illustrated by *Figure 12.2* which includes detection, diagnosis (or localization), and correction/repair operations.

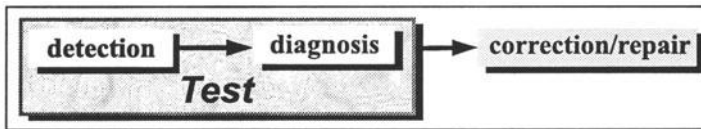


Figure 12.2. Test mechanism: detection - diagnostic - correction

Finally, the application of a detection or diagnosis test is generally carried out on a product disconnected from the process to which it is normally linked during the operation stage. Therefore, we say that this is **off-line testing**. This is not the only way to proceed, and we will meet other techniques which test products during their functioning in their natural environment: this other test approach will be referred to as **on-line testing**. It will be examined in Chapter 16, as it is often the first step of fault recovery in fault-tolerant systems.

12.1.2 Different Kinds of Tests and Testers

12.1.2.1 Test Equipment

The basic context of the test of electronic circuits is given by *Figure 12.3*. The circuit, called the **device under test** (or **DUT**), is connected to an external entity called the **tester**, or **test equipment**. This tester applies a **test sequence** to the product and observes how the product reacts. According to the circumstances, the tester is a human operator, a physical system or a software system.

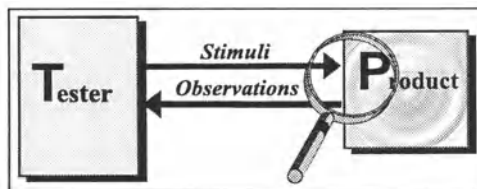


Figure 12.3. The tester

This situation correspond to *off-chip testing*, because the resources are allocated externally. On the contrary, *on-chip testing* uses embedded test resources. We will examine this new and developing approach when studying the *Built-In Self-Testing* (BIST) techniques (in Chapter 14).

12.1.2.2 Test Variety

Investigation techniques allowing a circuit to be tested are numerous. We can act on the circuit and observe the results by contact probing (electrical, and mechanical), or non-contact probing (electron-beam or laser). Investigation means can be *internal* (special test connectors and pads, scanning by electron-beam or laser) or *external* (normal input/output pins or special pins used for testing purpose only).

In electronics, we often meet the word ‘test’ in numerous different situations. In order to show this diversity, we take the particular case of the production of electronic equipment used in an aeronautic application. This equipment comprises integrated circuits designed and manufactured by a semiconductor manufacturer. *Figure 12.4* illustrates the different stages of this particular cycle and the verification and quality assurance actions conducted by the various agents involved in this process. These tests can be useful in fault prevention as well as in fault removal. We have already noted the tight links existing between these two dependability approaches.

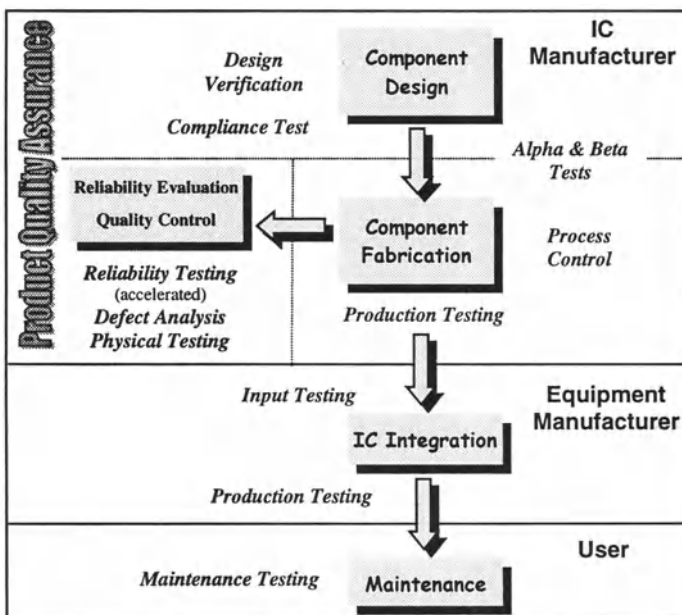


Figure 12.4. Examples of different test operations

These verification operations are firstly carried out during the design stages of the components considered here. These operations have already been presented in Chapter 10. Before the final commercialization, the components are subjected to *compliance test* to ensure their adequacy to the specifications. Then, these components are released to a “friendly” user (*Alpha test*), then, the scope of their use is enlarged to a still controlled group of users (*Beta test*). Additional *conformity tests* can be processed by external organizations or clients.

The IC manufacturer subject manufactured components and production machines to a certain number of tests during the manufacturing process and after manufacturing to ensure that the components will be of good quality and reliability. On the one hand, these operations concern the *reliability evaluation* and the *quality control* (by means of reliability accelerated testing, defect analysis, physical testing), and the *process characterization and control* already introduced in Chapters 7 and 11, regarding fault prevention. On the other hand, we meet the *production testing*, which is analyzed in this chapter. All the tests carried out in the normal manufacturing chain are applied to the majority of products, if not to all of them, and they have to be *non-destructive*. On the contrary, the quality control tests, and reliability evaluation tests are applied to significant samples of the components. They can be *destructive*, notably when we want to identify the fault at the origin of an observed failure.

The manufacturer of the final equipment integrates the components after having subjected them to tests specific of the application. As for IC production, compliance, alpha, beta, and conformity test procedures can be applied to the produced equipment. Then, the manufacturer naturally carries out a *production test* of his/her final equipment. Finally, a *maintenance test* is applied to the equipment during its active life, for example by the airline company using the equipment.

We complete this presentation in the following sections by considering only production and maintenance testing.

12.1.1.2.3 Production Testing

The production tests of the electronic components are the responsibility of the manufacturer of the semiconductors.

We essentially meet four types of such tests and therefore technological investigation means illustrated by *Figure 12.5*: the *parametric test*, the *continuity test*, the *logical* or *functional test*, and a group of diverse techniques called here ‘others’.

1. The *parametric test* considers the electrical aspects of the circuit, according to various power and load conditions: supply, drive and leakage currents through the pins, impedance values, noise immunity, and dynamic

aspects of component switching. For example, we measure the currents and voltages of a sequence of input/output signals, and we draw waveforms which are then compared to those of the data sheets. From a dynamic point of view, we can analyze the functioning of a component at different temperatures (-55°C, 25°C, 125 °C). The *IDDQ testing* measures the supply current of CMOS circuits. Defect-free CMOS circuits have very low levels of current during quiescent states. On the contrary, these current levels are higher in the presence of a silicon defect. So, IDDQ testing detects the physical defects that create a conduction path from the power supply to ground, hence producing an excessive current.

2. The *continuity test* ensures that the electrical links between the components are correct: chips and their packages, printed circuit boards, motherboards, and various kinds of cables and connectors.

3. The *logical test* (also called *functional test*) checks the logical function which has to be ensured by the circuit; this is the type of test which is the main subject of this chapter.

3. Other types of tests can also be applied to components: visual inspection, mechanical pin tests, corrosion tests, etc.

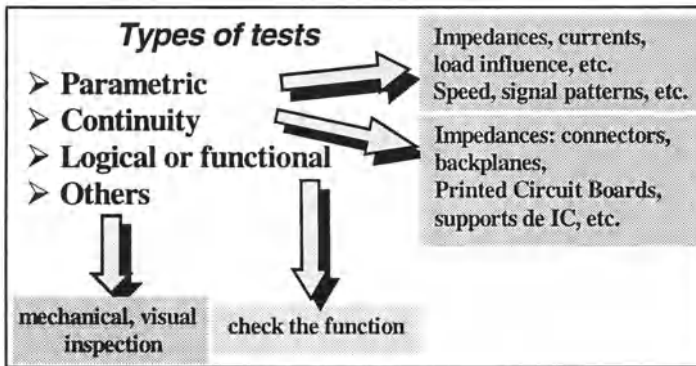


Figure 12.5. The four categories of production testing

Some of these tests are sometimes carried out with environmental constraints such as the temperature or the electric supply; however, these are *non-destructive burn in tests*. Figure 12.6 gives an example of test cycle of an ASIC production. This test cycle integrates a screening for infant mortality by a 4 hours electrical burn-in at 125 °C and at elevated voltage.

Furthermore, we find the term *schmoo plots* which defines a logical test which is applied by making certain parameters vary, such as the power supply voltage and/or the signal frequency. From this experiment, the correct functioning domains are deduced.

The integrator of the electronic components used in the final product’s manufacturing also subjects these components to input (incoming) tests. These electric or logical tests are often of ‘burn-in’ type, in order to eliminate the weak components. Therefore, they enter in the category of fault prevention already considered in Chapter 11.

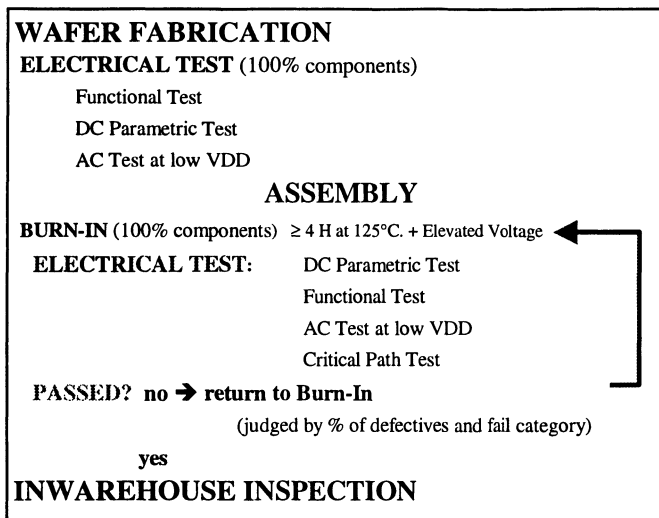


Figure 12.6. Example of test flow for ASIC devices

12.1.2.4 Test Equipment

Each of these test categories implies a very specific and complex *test equipment* (or *tester*) aiming at storing and processing information, driving (and extracting) signals to (and from) the tested product thanks to various circuits: bed of nails, high definition probes and electronic interfaces, switching matrices to propagate the signals, and so on. These testers are often very expensive: we can quote costs of testers around 5M\$! They are used in various production domains: electronic components, automatic systems, computing systems, communication systems and computer networks, etc. Certain of these testers are products manufactured and sold by specialized companies; others are specifically developed for particular applications (for example, proprietary Automatic Test Equipment in the aeronautical domain).

Of course, the complexity, costs and the importance of tests and testers have lead to the elaboration of standards for the description of test sequences and their application by the testers.

We quote the **STIL** standard (*Standard Tester Interface Language* normalized IEEE P1450) which is a language describing test patterns and

application protocols in standard generic form and the **IEEE P1500 (Embedded Core Test)** which is used for application of tests to embedded cores: test-description language, test-control mechanisms and peripheral access mechanisms.

Another example of the standardization of test equipment is the **VXI (VME eXtensions for Instrumentation)** which is a multi-vendor industry standard (IEEE 1155-1992). This standard is supported by numerous instrument and test equipment manufacturers (like Tektronix and HP). It permits the connection of numerous instruments such as waveform generators, analog-digital converters, relay matrices and drivers, digitizing counter/timers, specific simulation and fault injector modules, etc. It is notably employed in the automotive and avionics test application.

In the integrated circuit test domain, we find a large number of companies which manufacture specific testers:

- **Testers designed to address design verification and prototype test:** companies such as ASIX, Cadic, HP, HPL, Hilevel Technology, IMS, Tektronix, Texas Instruments, etc.
- **Production testers:** Advantest, Ando, Fujitsu, GenRad, Megatest, Mitsubtishi, Sentry, Tektronix, Teradyne, Toshiba, Trillium, etc.

12.1.2.5 Maintenance Testing

Some test production equipment and methods are employed for maintenance testing. However, the test objectives and constraints are generally different. In particular, the investigation means is more reduced than during production, and we are mainly interested in diagnosing the faults detected in order to proceed to repair them efficiently. These characteristics will be defined in the next section in relation to logical tests.

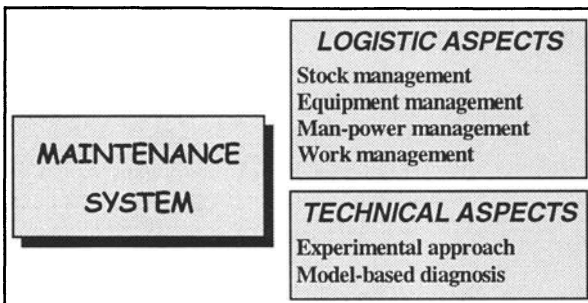


Figure 12.7. Computer Aided Maintenance

The maintenance calls for various operations: research of documentation, fault diagnosis, management of replacement parts, repair, and final system

test. Two categories of *Computer Aided Maintenance (CAM)* computing tools assist the maintenance team (Figure 12.7):

- *Logistics*: management of human resources, stock and budgets,
- *Technical*: maintenance techniques assisted by computer or systems which help the diagnosis.

Various classifications of the maintenance testing approaches have been proposed, according to the point of view considered: knowledge about the system and/or the faults and failures, type of information (certain or statistical), type of involved techniques (inductive or deductive), etc. We adopt a simple classification based on two groups: the *experimental approaches*, and the *model-based approaches*.

Experimental approaches

The *experimental approaches*, also called *empirical associations*, are based on knowledge of faults or errors, failures and their relationships. These relationships between observed symptoms (failures) and the list of possible causes (faults/errors), and sometimes tests to be executed to precise the actual cause, are stored in a *knowledge database*.

When a failure occurs, an *expert system* or a human operator determines the real causes using the database. As this analysis does not require any deep knowledge on the product behavior or structure, it is called *surface* or *shallow reasoning*. The term *reasoning by association* is also used.

The main problem with this approach deals with the definition of the knowledge database. It is constituted from two sources of information. Firstly, it uses experimental feedbacks from exploitation: the maintenance agents communicate information about real failures and their causes found by their own analysis. Secondly, this knowledge is produced by an analysis of the product before its use. This analysis involves two kinds of methods:

- *Inductive methods* such as the *FMEA*. Starting from supposed fault (a fault model is supposed known), the induced failures can be determined.
- *Deductive methods*, such as the *Fault tree Method (FTM)*. Starting from imagined failures, their causes as faults or errors are determined.

In practice, the two methods are used together: the database contains initial pieces of information obtained by an analysis of the system; this basic knowledge is then extended by experimental feedback from operation.

Model-based approaches

The *model-based approaches* do not explicitly assume any fault or error model. Hence, no relationships between failures and faults exist a priori.

These relationships are established for each failure to be diagnosed, using the system modeling: specified behavior, designed structure, etc.

The occurrence of a failure and the determination of the causes are established by a comparison of the modeling of the actual behavior or structure and the corresponding modeling of the faultless system. For this reason, this approach is also called *diagnosis based on deep knowledge*, or *diagnosis based on structure and function*.

To make the diagnosis easier, a *diagnosis algorithm*, or *diagnosis process*, is provided. It defines firstly, modeling tools, such as the diagnosis fault tree introduced in sub-section 12.2.4. It defines secondly, tasks or steps, an example of which will be given in section 13.7 of Chapter 13 dealing with structural testing, as this diagnosis method needs knowledge on the system structure. Certain tasks can use for instance the application of fixed or adaptive test sequences.

To conclude, *experimental approaches* are mainly used when associations between fault/error models and failure models, such as in Electronics. The *model-based approaches* are efficient when such associations cannot be provided as database. The term *experimental approaches* refers to established knowledge, and does not imply additional experiments on the product, such as test sequences which can be used in *model-based approaches*.

We must also mention that the techniques based on these approaches can be used *off-line*, that is after a failure occurrence in a system which is stopped, or *on-line*, that is during operation (*self-diagnosis*). In practice, both approaches are often used together, and are partially implemented in the system (for instance, error detection only), and handled off-line.

12.2 LOGICAL TESTING

We will now focus on the *logical test* issues during the production and maintenance stages.

12.2.1 Logical Testers

12.2.1.1 The Three Basic Families of Testers

In order to simplify our study, we suppose:

- that the test has one single sequence (called *input sequence*) constituted of n input vectors applied to the product ($\langle e_i \rangle$),
- and that the tested product responds to this sequence by an *output sequence* of n output vectors ($\langle s_i \rangle$).

Each vector e_i is called elementary *test vector*. The *test sequence* is the list of all test vectors. Sometimes, the test sequence represents the input sequence only. The test can be constituted of several sequences with or without previous product initialization, and their length (number of vectors) is not necessarily the same. We should note that numerous products do not satisfy the simplistic rule of '1 input vector - 1 output vector'. For example a microprocessor does not respond immediately to each applied input vector. To deal with this case, we introduce the notion of 'no value' as one of the normal output values. In addition, the tester may only look at output data at certain times during a period; this is usually called *strobing* technique. Thus, we do not observe the outputs in a continuous manner. For example, when a test vector is applied to a microprocessor, it can provoke a sequence of internal operations (it is the case when a microprocessor executes an instruction); then, the output data given by the microprocessor is sampled at the end of this sequence (hence, after a certain number of clock pulses).

Three different logical test approaches exist: with a *reference list* (illustrated by Figure 12.8-a), with a *standard* (or *referent*) product (Figure 12.8-b), and by *signature analysis* (Figure 12.9).

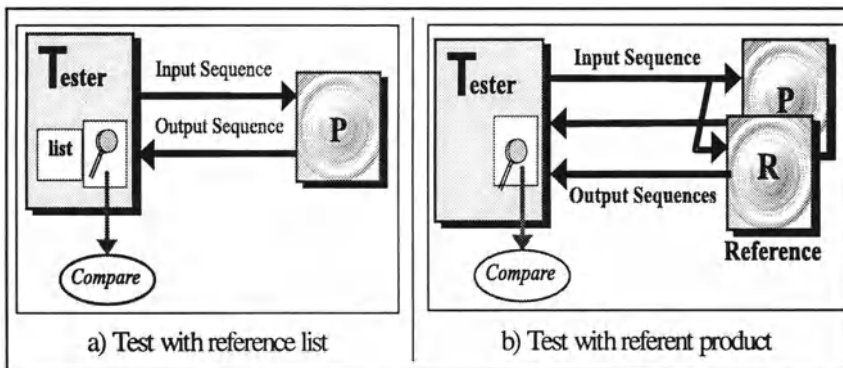


Figure 12.8. Test with reference list and referent product

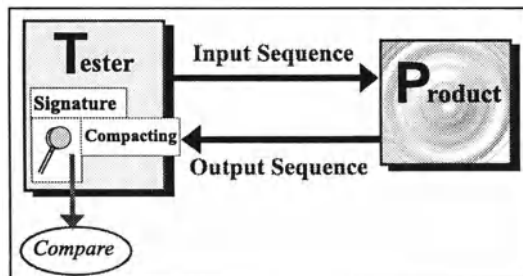


Figure 12.9. Test by signature

12.2.1.2 Test with Reference List

The tester applies a test sequence and compares the output sequence delivered by the product with a predefined *reference list* (stored in the tester's memory). This is known as a *determinist* test whose sequence is obtained either automatically (algorithmic test generation) or manually (thanks to the designer's 'know-how'). This approach is the most efficient, but it is not the most used due to the difficulty in determining the test sequence; this has been mentioned in Chapter 10 for design verification.

12.2.1.3 Test with Referent Product

The tester simultaneously applies the input vectors to the tested product and to a *standard product* (or *referent product*) which is supposedly fault free. The comparison between the two obtained output sequences allows the tester to decide if the tested product functions correctly. The applied test sequence is often *random*, or *pseudo-random*. The random aspects of these sequences can also be constrained by information about the product's functioning or structure: for example, we choose the inputs which are the most frequently applied to the product during its useful life, or subsequences allowing to enter in a buried part of a multi-level sequential system (case of a microprocessor), etc.

The expected outputs do not have to be known a priori, contrary to the previous approach. This technique is greatly used, as it is simple to carry out. The advantage of not requiring knowledge about the expected outputs is important. In Chapter 10 we have already explained the difficulty to obtain these output values in some cases. However, this technique is based on the trust that we can bring to the standard referent product. Moreover, it is not always possible to dispose of a reference exemplar of the tested product.

12.2.1.4 Test by Signature Analysis

With *the test by signature analysis*, the tester does not have any precise reference about the expected product's output values. It treats these values by a reducing mathematical transformation (*compaction*) to extract a *signature* whose likelihood establishes the correctness of the product's functioning. This technique originates in the test by observation of characteristics of some signals of analog electronic equipment. Thus, to test a television set, we can examine with an oscilloscope the waveforms of signals present in certain predefined places, in order to detect certain faults.

This technique offers the advantage of being able to reduce the length of the output sequence to analyze. The complexity of the tester is drastically reduced. This is why this approach develops today in the frame of integrated test *Built-In Self Test* (BIST) examined in Chapter 14.

This technique applies also to software systems. The signature is defined by *property* on some data. When the property is true, the program is considered as correct, and when the property is false, the program is said to be incorrect. For instance, the type of an output parameter defines a property: the output values must belong to the range specified by this type.

The main drawback of signature analysis approach is a sometime large limitation in the fault coverage. Indeed, the satisfaction of the defined property does not guarantee the absence of faults in the product. For example, let us consider a product regulating the temperature of a car radiator. The type is an integer included in the range [0, 100]. If a fault blocks the data delivered by the sensor at 20°C, the sampled value satisfies the type property, while the actual value is 105°C, leading to a failure of this regulation product.

12.2.2 Test Parameters

The main test issue is the determination of the *test sequence*. Several parameters allow choosing test methods appropriate to assigned objectives and to existing or affordable means. These parameters characterize:

- the test sequence generation: *facility of the generation* of the test sequence, *cost* of the *associated means* (humans and tools),
- the *test sequence quality*: its *length* or number of test vectors (which conditions the test duration), and its efficiency in terms of *fault coverage rate* (that is to say the percentage of faults revealed by the test).

As previously mentioned, two distinct missions can be assigned to the tester: *detection* and *diagnosis*. The detection test identifies the product's state as 'good' or 'bad'. The diagnosis test refines the analysis and determines the elements affected by faults. Hence, it is more complex: the test sequences are much longer, and their obtaining is more difficult.

The test sequences are determined by automatic or manual *test generation* methods. These methods can be split into two distinct groups:

- *with fault model*, when we try to detect the effects of faults and to diagnose the faults relevant to a given model,
- *without fault model*, when no precise hypothesis is made on the type of faults considered.

The notions of faults associated with a technology have been introduced and explained in Chapter 5. We should remember that the traditional term 'fault model' in reality often covers classes of errors (regrouping classes of faults) called *error models*. Thus, the basic fault model for logical circuits is the single stuck-at '0' or at '1' faults of the inputs and outputs of the logical

gates which constitute the circuit studied.

The methods of the first group often apply to systems, whose detailed structures are known, for example, the electronic circuits whose implementation technology is mastered by the manufacturer. We can therefore judge the quality of the test (*test evaluation*) by the *coverage* criteria, which is relative to a precise fault model.

The *fault coverage* of a test sequence is the ratio between the number of faults detected and the total number of faults contained in the fault model.

For example, if we consider the previous simple stuck-at 0/1 fault model of a logical circuit with a total of w gate inputs and outputs, the coverage of a test detecting d faults is $c = d / (2.w)$; each line can be stuck either at '0' or at '1'.

On the contrary, the second approach (without fault hypothesis) concerns the products for which we do not know any precise and representative fault model; this is frequently the case with software. The test sequences are therefore not determined from faults which can affect the structure, but from specifications which define the expected behavior. We will discuss this with regard to the efficiency of such sequences in section 12.3.1.2.

Some logical test methods with fault models for hardware and software products will be introduced in Chapter 13. Again, a complete and detailed presentation of the hardware and software test methods would necessitate an entire book.

The off-line test therefore concerns the production and utilization stages. In these two stages, the problems raised and the solutions used are not exactly identical. We are going to successively examine the *production test* associated with the production phase, then the *maintenance test* associated with the operation phase.

12.2.3 Production Testing

As already noted, the *production test* essentially concerns the production of hardware systems (complete circuits or equipment). Due to the constraints imposed by the production rates of electronic components, the manufacturer generally applies a detection test which is as short as possible. This is a test with fault model obtained using a *fixed sequence*. The expected input vectors e_i and output vectors s_i are known. We note $v_i = (e_i, s_i)$. The n vectors of the sequence are applied one per one, and each time we compare the output delivered by the product with the expected output. The product is reputed to be 'good' when it passes the n vectors with success. On the contrary, it is reputed to be bad as soon as it provides a different response

than the expected value. This test, sometimes called **GO-NOGO**, provides therefore two sets of products: those qualified as ‘good’ and those qualified as ‘bad’ (see *Figure 12.10*).

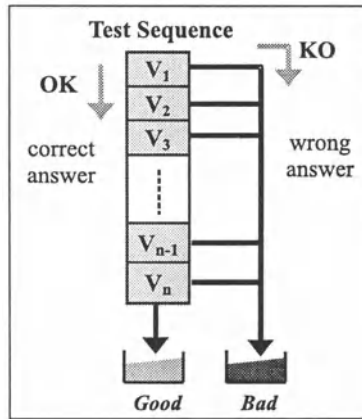


Figure 12.10. Detection testing

Let us note that a different approach could be taken to test with **adaptive sequences**, which are defined dynamically, taking the previous results of the test into account. The adaptive testing is essentially used for diagnosis and will be discussed in sub-section 12.2.4.3.

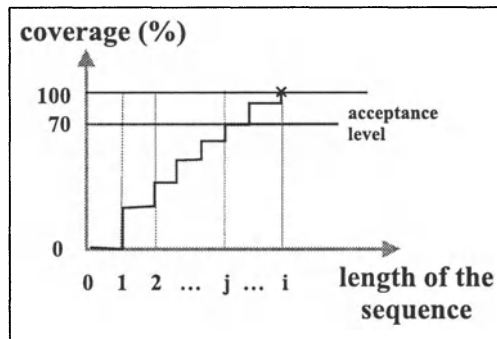


Figure 12.11. Fault coverage

For a production, the **yield** is the statistical percentage of good products on the total number of manufactured products. The average duration of the test depends on the proportion of defective manufactured products.

Production constraints lead electronic component manufacturers to demand test sequences with limited duration (e.g., a component’s test should not go over 10 seconds). Then, there is a (negotiated) reduction of the fault

coverage which may become inferior to 100%. *Figure 12.11* shows that the sequence length can be reduced from i (optimal value of the test at 100%) to $j < i$ vectors, if we accept a reduced coverage to 70% of faults.

Consequently, the product buyer has to be very careful and verify:

- the number of components which are effectively tested (we speak here of test at 100% when all the components leaving production are tested),
- the fault coverage of the applied test,
- and the fault model considered.

Example 12.1 illustrates the notion of efficiency of a test sequence according to the fault coverage. Example 12.2 analyzes the influence of the fault coverage on the production yield.

Note. The length of the test sequence, and therefore the cost of the test applied to the product, can be one of the reasons justifying big differences in component prices. A component from the same production chain could have a higher price than another one, due to the fact that it has been subjected to a deeper test. Therefore, the client pays for the *justified supplementary trust* he/she can place in this product, that is to say its dependability.

Example 12.1. Test coverage of a NAND gate

Let us consider a 3-input NAND gate (*Figure 12.12*) and a ‘single stuck-at 0/1’ fault model. We firstly analyze the *exhaustive test sequence* constituted of the 8 input vectors, from 000 to 111. Each input vector tests some of the 2x4 single stuck-at 0/1 faults.

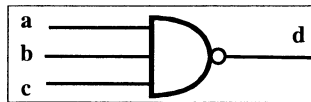


Figure 12.12. Three-input NAND gate

The **fault table** (or **coverage table**) of *Table 12.1* shows all the faults detected by the 8 input vectors: on each row, the symbols 0, 1, and -, respectively indicate the detection of a stuck-at 0, a stuck-at 1, or no detection of the corresponding line (columns a , b , c , and d). For example, if the input vector (011) is applied, we detect the stuck-at 1 of input a , and the stuck-at 0 of output d , as the output equals 0 instead of the expected 1.

It should be noted that, for the moment, the determination of faults detected by a vector is done by a simple comparative analysis of the functioning with and without faults. More efficient methods will be presented in the next chapter.

Input vectors a b c	Test coverage			
	a	b	c	d
0 0 0	-	-	-	0
0 0 1	-	-	-	0
0 1 0	-	-	-	0
0 1 1	1	-	-	0
1 0 0	-	-	-	0
1 0 1	-	1	-	0
1 1 0	-	-	1	0
1 1 1	0	0	0	1

Table 12.1. Fault table

From this table, we can deduce the coverage curve corresponding to any given test sequence. For example, the left part of *Figure 12.13* shows the coverage curve of the exhaustive sequence. The first vector (000) detects one fault of the 8 possible faults. The following vectors (001 and 010) testing exactly the same fault, the coverage is not improved: it remains at 1/8, We must wait for the last test vector (111) to reach a 100% fault coverage.

Let us now examine the optimal test sequence which has 4 test vectors: <011, 101, 110, 111> (the determination of this optimal test sequence will be considered in Chapter 13). The second part of *Figure 12.13* shows that each vector in this sequence brings the detection of new faults.

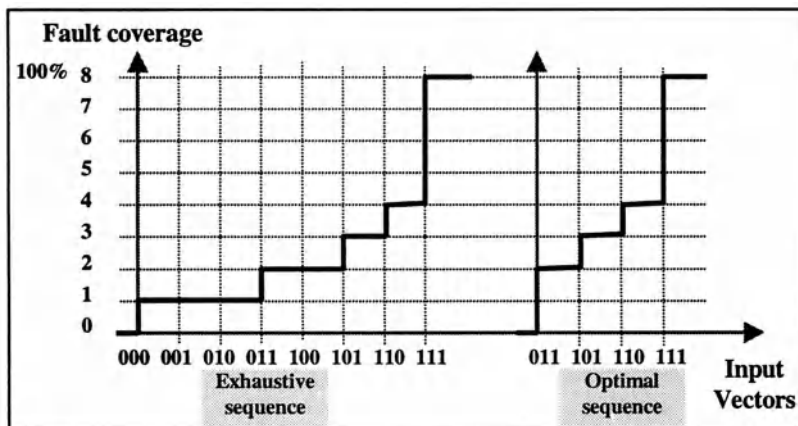


Figure 12.13. Fault coverage of two test sequences

Example 12.2. Production test coverage vs. yield

The test of the components of a given integrated circuits manufacturing chain has the following characteristics:

- all the components are tested with a fixed test sequence,
- the fault model is the set of the permanent stuck-at '0/1' of the inputs/outputs of the logical gates,
- the sequence covers 70% of these circuit's faults: $c = 70\%$.

From this data we deduce that, if the statistical yield of this production is $y = 95\%$, a probability exists $p = (1 - c).(1 - y) = 0,3 \times 0,05 = 0,015$ that a component affected by a fault is however declared as 'good' when the test is issued. This result is the number of statistically defective components whose faults have not been detected. Exercise 12.3 comes back to this study.

12.2.4 Maintenance Testing

12.2.4.1 Corrective, Preventive, and Evolutive Maintenance

We have already pointed out in Chapter 7 that the *maintenance* is an operation integrated into the utilization stage of the product: the product's functioning is stopped for a test, and eventually, a correction or repair action is made if a fault is detected and if the product is repairable.

The decision to carry out maintenance operations depends on the product use. In certain cases, we decide to test the product because a failure has been noticed during its operation. For instance, we ask for repairing a household appliance or a hi-fi equipment which is defective. This is known as *corrective maintenance*. On the other hand, in other cases, a policy of systematic and periodic maintenance is practiced, even if the product does not show any sign of failure. This is the case for example in avionics where the equipment is checked systematically with strict and predefined scheduling after a given period expressed in number of flight hours. This is known as *systematic preventive maintenance*. The maintenance action can also be decided after the occurrence of some events detected on-line, for example when the temperature becomes excessive. This case corresponds to *conditional preventive maintenance*.

Maintenance often has a far wider meaning than just a simple test, notably in the software domain. This implies operations which tend to improve or make the product evolve according to new constraints (e.g. increased performance) or new functionality (e.g. offering new services). This is known as *evolutive maintenance*.

12.2.4.2 . Detection and Diagnosis

Everything that has been said in the previous paragraph regarding manufacturing tests is of course valuable in the case of maintenance tests. However, the detection requirements (measured for example in terms of fault coverage) are often greater whilst the time available to test is also greater.

In addition, it is often necessary to push the investigation further in order to diagnose the fault or faults which affect the *repairable* products, before repairing them. This is known as *diagnosis* (or *localization*) *testing*.

The diagnosis testing supposes knowledge about a more or less refined fault model, according to the nature of desired investigation: MOS transistor level, IC level, PCB level, etc.

This kind of test is more complex to obtain and its duration is much longer than the detection test. Indeed, this test must not only signal failure but also provide information which allows the internal cause to be deduced. The detection test tries to cover a maximum number of faults with the minimum number of test vectors. In the case of Example 12.1, the vector (111) covered four faults (stuck-at 0 of lines *a*, *b*, and *c*, and stuck-at 1 of line *d*). All faults detected by an input vector are said to be *pattern equivalent faults*, that is faults provoking the same failure. When dealing with diagnosis test, we want to split these faults into separate classes. For example, is it possible to distinguish between the 4 previous faults? Unfortunately, as we will study latter, some faults cannot be distinguished from the outside of a circuit: such faults are said to be *system equivalent faults*. This is the main issue of diagnosis techniques.

Traditionally, and for reasons of operation duration, the maintenance test has two successive stages:

- *detection testing*, which allows the ‘good’ or ‘bad’ functioning of the product to be known rapidly,
- *diagnosis testing* applied if the previous test reveals a failure, in order to find the faults responsible and then correct them.

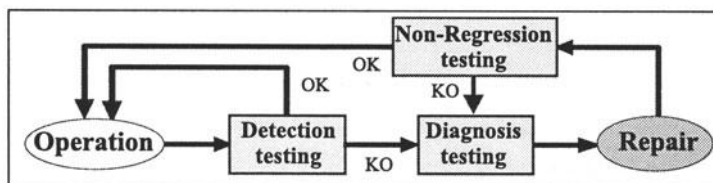


Figure 12.14. Maintenance testing

When the diagnosis test is made, the products which present faults are repaired by technicians who change the defective components. After this

repair, it is advised to re-apply a detection test in order to ensure that the repair has not introduced new faults. This test is called a *non-regression test*. Figure 12.14 illustrates the complete maintenance cycle which presents the four previous operations.

12.2.4.3 Fixed and Adaptive Diagnosis Testing

Two different test categories allow fault diagnosis:

- *fixed diagnosis testing* illustrated by Figure 12.15 a),
- *adaptive diagnosis testing* illustrated by Figure 12.15 b).

The *fixed diagnosis testing* records all the product’s responses (output values) in the form of a vector, called *signature*, containing n elements. Each element of this vector corresponds to the response of one test vector: if the product’s outputs are correct, we write ‘good’ (noted OK), if not we write ‘bad’ (noted KO), coding when possible the different forms of incorrect responses. If the signature contains ‘good’ elements only, the product is correct. On the contrary, it is necessary to analyze the signature to localize the fault or faults with the help of a *diagnosis tree* that will be described in sub-section 12.2.4.4.

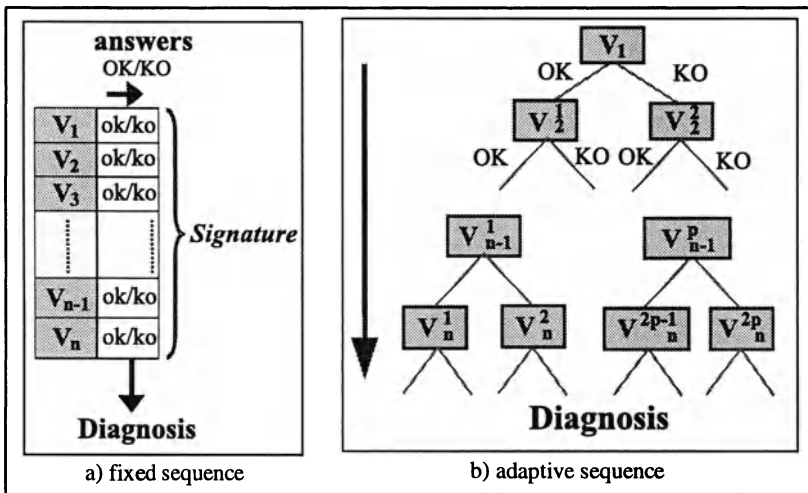


Figure 12.15. Diagnosis testing

The fixed diagnosis testing is often questionable. To illustrate the criticisms, imagine a doctor who asks his/her patient a fixed list of predefined questions. Such a diagnosis method can lead to questions that are irrelevant, or without any answer. In reality, a doctor adapts his questions to the previous patient’s answers to obtain rapidly a precise diagnosis. This

technique is known as *adaptive diagnosis testing*. The tester reacts to the product's good or bad response at each applied test vector: the next vector depends on this response. This approach is more efficient in terms of test length, as the diagnosis tree is developed as the test progresses, but the test sequence is a lot more difficult to elaborate.

12.2.4.4 Diagnosis and Fault Tree

Basic principles

In this sub-section, we consider a product, a fault model, and a test sequence. The first test vector of this sequence detects a sub-set of the fault model. Before the application of this pattern, either the product is faultless either one of the fault of the fault model is present. We note F the fault set plus the fault free case. The application of the test to the product splits this initial F set of the model into two complementary classes: faults detected by this vector (noted $D1$ in *Figure 12.16-a*) when the product gives an incorrect answer (noted KO, the test 'fails'), and faults not detected by the vector, or faultless product (noted $D1'$ in *Figure 12.16-a*), in the opposite case (noted OK, the test 'passes'). For the moment, we suppose that an incorrect answer given by the product does not allow distinguishing between these non-detected faults. After these two test vectors are applied, the faults detected are the union of the two sub-sets $D1$ and $D2$ (*Figure 12.16-b* and *-c*). The coverage of a test sequence is 100% if all the faults belong to at least one set D_i . However, this test sequence execution cannot identify the existing fault.

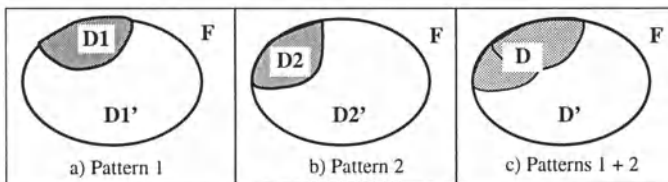


Figure 12.16. Fault detection

The *diagnosis fault tree technique* extends the previous approach by using the pieces of information provided by the fault partition made by all the vectors. Hence, after application of the two first test vectors, the F set is split into 4 sub-sets (see *Figure 12.17*) according to the consecutive results of the two tests:

- $D1 \cap D2$ for a test result $\langle \text{KO}, \text{KO} \rangle$ (the two tests failed),
- $D1 \cap D2'$ for a test result $\langle \text{KO}, \text{OK} \rangle$ (the first test failed and the second one passed),

- $D1' \cap D2$ for a test result $\langle \text{OK}, \text{KO} \rangle$ (the first test passed and the second one failed),
- $D1' \cap D2'$ for a test result $\langle \text{OK}, \text{OK} \rangle$ (both test passed: no fault revealed).

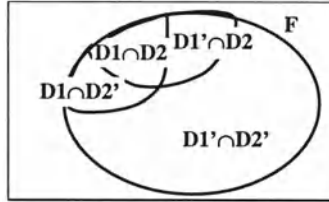


Figure 12.17. Fault partition

These four cases can be represented with a *diagnosis tree* (Figure 12.18), which is here a binary tree. If a leaf of this tree contains only one fault, the signature of the test answer leading to this leaf identifies this fault. On the contrary, if a leaf contains several faults, they are said to be *pattern equivalent*, i.e. relatively to the test sequence. Another test vector might split these faults. If no test vector can separate this sub-set into smaller parts, these faults are said *system equivalent* according to the external controllability and observability. A diagnosis sequence is a **complete distinguishing sequence** if all resulting faults classes contain *system equivalent faults*.

Notes. Sometimes, some parts of the diagnosis tree are ‘impossible’, that is to say, some sub-set intersections are empty. Consequently, a diagnosis tree can reveal ‘impossible’ leaves. We will encounter such cases later.

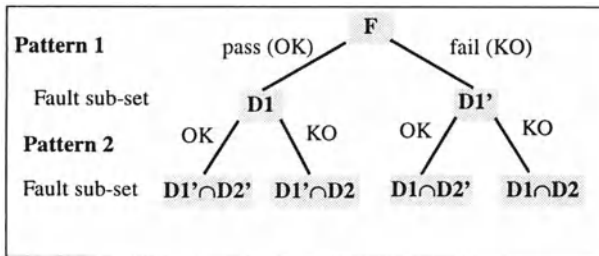


Figure 12.18. Fault tree

Example 12.3. Diagnosis of a AND gate

Let us consider the 2-input AND gate represented in Figure 12.19.

If the $ab = 01$ vector is applied, the normal output equals ‘0’; observing a ‘1’ at the output can result from a stuck-at ‘1’ fault of the a input or a stuck-at ‘1’ of the output c . In order to know which one of these two faults is

present, a second test vector must be applied, for example '10': if the output again presents an error (value '1') the fault affects *c*, if not it affects *a*. We note that it is impossible to distinguish the stuck-at '0' faults of *a*, *b* and *c*, as the output has the same value no matter the vector applied at the input.



Figure 12.19. AND gate

General diagnosis tree

Whatever the type of diagnosis used, whether *fixed* or *adaptive*, the test sequence carries out a *partition* of the considered model's faults into several distinct classes. The diagnosis sequence is complete if each class contains a single element or a group of system equivalent faults. Till now, we supposed a simple OK/KO answer to each test vector. In the general case, more complex situations may occur, with several classes of bad answers to each test vector. The combination of all the situations corresponding to the applied test vectors develops in the form of a tree which possesses as many layers as there are test vectors. Each node, (application of a test vector), has as many outgoing arcs as there are possible answers.

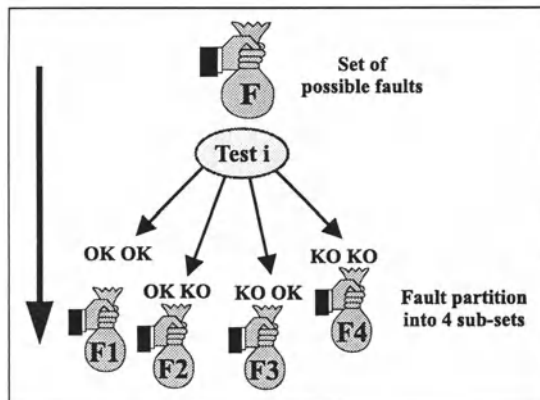


Figure 12.20. Fault partition implied by a test vector

Thus, for a circuit which has two outputs *z1* and *z2*, four cases are possible during the application of any test vector (Figure 12.20):

- the two outputs are correct (situation noted as 'OK OK' in the figure),
- the first output *z1* is good and the second *z2* is erroneous ('OK KO'),

- the first output $z1$ is erroneous and the second is good ('KO OK'),
- the two outputs $z1$ and $z2$ are erroneous ('KO KO').

Each of these responses corresponds to disjoint situations: disjoint event sets, with or without the presence of faults. This operation is illustrated in *Figure 12.20* by a set partitioning: the set F of established possibilities before the application of test i is split into 4 smaller sets ($F_j, j = 1, \dots, 4$) according to the product's response to the test vector. The union of these resulting sets reconstitutes the initial set F .

The successive application of n test vectors therefore partition all the initial possibilities (no fault or each one of the fault model) into increasingly reduced different classes. We will analyze an example in Chapter 13.

12.3 PRINCIPLES OF LOGICAL TEST GENERATION

This section establishes the principles of the determination of test sequences for logic circuits modeled with gates. We first expose in sub-section 12.3.1 the general problems of the logical test. Then, in sub-section 12.3.2 we propose an intuitive test generation method, based on *path sensitizing*, to find the vectors detecting a fault of a combinational circuit described by gates. In sub-section 12.3.3 we present methods allowing to evaluate the fault coverage of test sequences (*fault grading*); they are complementary to the test sequence generation methods. Sub-section 12.3.4 synthesizes the two previous approaches to define an algorithm for *automatic test pattern generation*. Finally, in sub-section 12.3.5 we discuss the problem of *sequential circuit* testing.

12.3.1 Logical Testing

12.3.1.1 Main Approaches

The aim of this section is to introduce the different approaches used to determine the test sequences, and to show their relative difficulties. Whether automatic or manual, *test sequence generation* is an operation which is often complex, even for combinational logic systems (without memory). Numerous methods of test sequence generation have been imagined and applied. They can be ordered according to several criteria:

- The modeling level of the product to test:
 - functional level,
 - structural level,

➤ structured-functional level.

- The refinement level of the fault model employed (the absence of fault model being a limit case).
- The method to determine the test sequence: algorithmic, random, etc.

From a practical point of view, five approaches deserve our attention:

- *Exhaustive test* (without accurate fault model),
- *Functional test* (without accurate fault model),
- *Toggle test* (without accurate fault model),
- *Random and pseudo-random test* (without accurate fault model),
- *Algorithmic structural test* (with fault model).

The *exhaustive test* is a systematic approach. It consists in applying all possible vectors of the input domain! Therefore, this is a 2^n -vector sequence, which increases in an exponential manner to test a combinational circuit with n inputs. This type of test is only interesting for circuits having a reduced number of inputs. In the case of sequential circuits, that is to say circuits whose behavior depends on an internal state, the sequence can be prohibitive, as all the state values must be taken into account. For example, the test of a simple 32-bit counter would require more than one hour with elementary tests of $1\mu\text{s}$.

The *functional test* already evoked for detecting functional faults during the design stage (Chapter 10) is the universal test tool, whatever the nature of the product. If, for example, the circuit is an adder, we would execute the addition of two numbers and we would compare the result with the exact value of the calculation. The structural model and the fault models are not used. Universally and intensively used, this simple method is far from satisfying the specialists, as the fault coverage is unknown.

The *Toggle Test* is a variant of the simulation which analyzes the structure and seeks to make each component, line or variable evolve in all its states: for example, each line will have at least once the value '0' and the value '1'. With regard to the simulation, the improvement comes from the structure's exploitation and the *activation* of the components. However, we can show that the *Toggle Test* does not guarantee 100% coverage of the traditional hardware fault models in electronics (this will be done in Example 12.4).

Random and pseudo-random test select test patterns randomly, or by using some heuristic, and use fault simulation to determine the faults detected by each vector. Test vectors are selected and added to the test sequence if they detect any previously undetected faults. The test generation

process is stopped when some required fault coverage level or computation time limit is reached. This method finds test patterns for the easy-to-detect faults quickly, but it becomes less and less efficient as faults are removed from the fault list and only the hard-to-detect faults remain.

In the *algorithmic approach*, a specific *Automatic Test Pattern Generation (ATPG)* algorithm is used to generate a test for each fault of a fault model associated with a structural model of the circuit. Most of the ATPG algorithms can be proven to be complete; that is to say, they are guaranteed to find a test for a fault, as long as such a test exists. This is the only efficient approach to reach high fault coverage rates.

Note. An efficient combined method for solving the ATPG problem uses firstly statistical methods to find test vectors for the ‘easy-to-detect’ faults on the fault list, and then switches to an algorithmic method to find test vectors for the remaining ‘hard-to-detect’ faults.

12.3.1.2 Functional and Structural Testing

Testing methods can be split into two groups, taking the considered system model into account: *functional* or *structural* model. Their comparative advantages and disadvantages are illustrated in *Figure 12.21* and discussed afterwards.

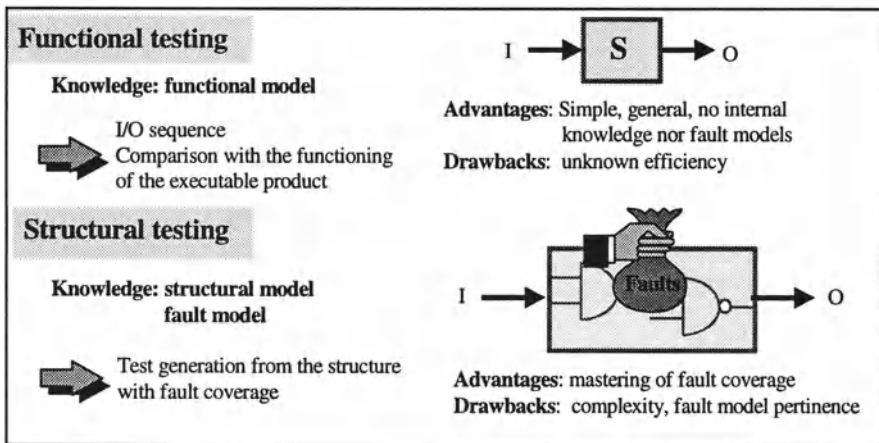


Figure 12.21. Functional and structural testing

The *functional test sequence* aims at processing all possible functioning cases of the system, and thus detects all possible faults by revealing their effects on the product behavior. However, only a sub-set of all possibilities are generally activated, assuming that they are representative of the whole functionality. For instance, all the values of a float parameter cannot be

exercised. So, the float domain is split into functioning classes and only one value of each class is used for test purpose. Unfortunately, it is possible that two different parts (circuits or programs) of the system structure are used to run two different values of the same class!. Thus, one part only will be exercised and not the second one which can contain many undetected faults. More generally, the efficiency of a functional test sequence concerning the detection of actual faults in the product is not known.

Thanks to the fault coverage notion, the efficiency of a *structural test sequence* can be assessed by a grade. However, the pertinence of the considered fault model must be discussed. Besides, the choice of a very refined fault model impacts the test sequence generation: high difficulty to obtain the test sequence which has generally a prohibitive length. For example, it is extremely difficult to master the test generation of a computer at MOS level! In conclusion, a compromise has to be found between the degree of the fault model's high precision and the test complexity (difficulty of obtaining the sequences and length of these sequences).

The production and maintenance tests are essentially faced with technological faults. The *structural* approach completes the *functional* approach introduced in Chapter 10 for the design test. In numerous cases, the test engineer starts with the functional verification sequences developed by the design engineer. Following this, he or she completes this sequence by a structural test approach.

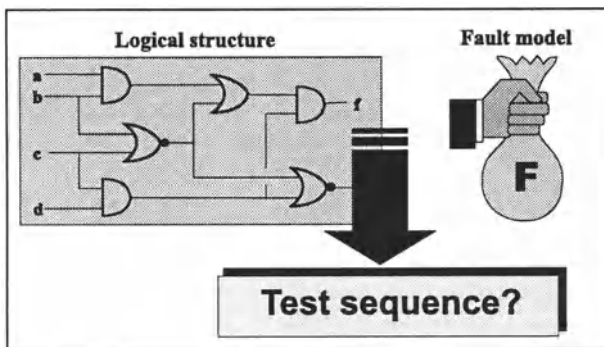


Figure 12.22. Context of structural logical test

In the following sub-sections we develop the problem of obtaining a structural test at the gate level. We consider a logical circuit structured into gates and a fault model which can affect it (Figure 12.22). We assume that each fault is a single stuck-at '0' or '1' of gate inputs and outputs. The tester applies certain values to the primary inputs (that is to say external to the circuit) and it observes the values produced at the primary outputs (that is to say external to the product). From this hypothesis, the detection test problem

consists in finding a sequence of input vectors which, by observing the primary outputs, allows the following question to be answered:

Is the circuit without fault, or is it affected by one of the fault model?

The length of the obtained test sequence is important, as it has consequences on the time spent to test a product. Unfortunately, the determination of minimal test sequences having a high coverage is intractable for complex circuits. Thus, we often accept reduced fault coverage, in order to obtain sequences of a reasonable length.

12.3.1.3 Fundamental Steps of the Test Process

The complete test development and application cycle has three steps as illustrated in *Figure 12.23*:

1. **test pattern generation**, using any method, manual or automatic (ATPG), algorithmic or random, with or without a fault model,
2. **test validation**, based on **test evaluation** (quantitative approach) or **fault grading**, which allows the efficiency of the previous sequence to be checked by using independent methods (generally by fault simulation) from those used for test generation,
3. **test application** with the help of a tester, as already mentioned.

The two first steps are frequently used in an iterative manner. The generation of a test sequence is therefore an incremental process which gradually builds the test sequence.

The following sub-sections enter into more detail about the test generation and fault grading methods of circuits at the gate level.

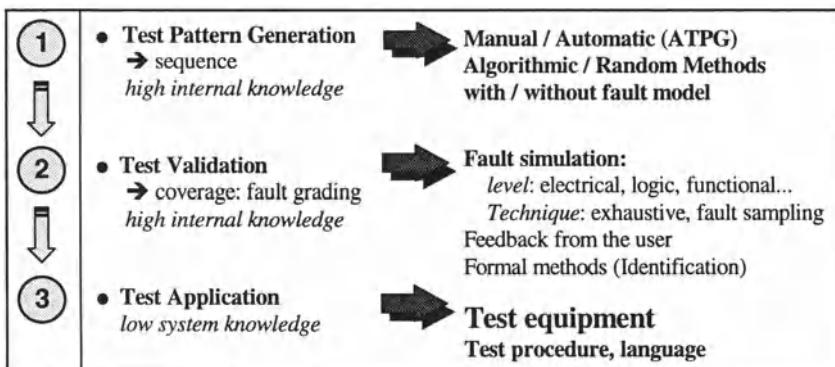


Figure 12.23. The three test steps

12.3.2 Determination of Input Vectors Testing a Fault

The goal is to find a set of test vectors which ensure the required fault coverage. It should be known that the generation of a test sequence of a combinational circuit is a 'NP complete' mathematical problem.

Amongst the structural methods of the detection test generation, the *path tracing approach* (or *path sensitizing*) is the best known. It allows vectors to be found which detects a given fault. This approach was popularized by IBM with the D-Algorithm proposed by J-P. Roth in the 1960s. It was then diversified and improved by numerous techniques such as Lasar (Logic Automated Stimulus And Response), Pódem (Path Oriented Decision Making), Fan, Tops, Socrates, etc.

What is remarkable with these path tracing methods is that, associated with design methods facilitating the test (examined in Chapter 14), they have been extensively used for computer testing for more than 30 years!

In Chapter 13 we will study an intuitive method based on path tracing. This method allows the fundamental problems of test sequence generation to be clearly understood, such as problems linked to fault controllability and observability in logical structures.

Whatever the method used, the research of circuit test vectors do not necessarily converge. Indeed, some circuit faults are not detectable externally to the product; hence, the product tolerates these faults which do not directly lead to a failure. In that case, the product has passive redundancy, already been described and illustrated in Chapter 8. Passive redundancy can result from a voluntary action such as the *Triplex* fault-tolerant structure (analyzed in Chapter 18), but it can also and very often be introduced in a totally unintentional manner by the product creation process. In Chapter 13 we will show some examples of the negative influence of passive redundancy on test detection and diagnosis.

12.3.3 Fault Grading

12.3.3.1 Principles

In the previous sub-section, we considered methods to obtain input vectors detecting given faults. *Fault grading* is an approach which aims at finding the coverage of a given input test sequence, that is, to evaluate this test sequence. Two main groups of methods have been proposed:

- methods by *structural analysis* which study the faultless circuit and deduce all the faults whose presence produces failures,
- methods by *fault simulation* which compare, by simulation, the faultless circuit's behavior with the circuit affected by faults of the fault model.

The techniques based on *structural analysis* determine all the faults detected by a given input vector (or sequence). Therefore, the fault coverage is deduced automatically. A simple approach will be discussed in detail in Chapter 13. It is based on a backward circuit analysis, from the primary outputs towards the primary inputs, in order to find all the detected faults. Even if the technique presented is specific to a gate model and a stuck-at fault model, its study is interesting to understand the general issues.

Fault simulation regroups a large variety of techniques based on structural simulation with faults injected from a fault model: the tools are called *fault simulators*. The principle of this approach, often called *fault injection*, is illustrated in *Figure 12.24*. It consists in applying the test sequence to be evaluated to a simulated model of the system studied, and injecting faults from the fault model. If the obtained output with a fault is different from the faultless circuit's output, this fault is noted as 'detectable'.

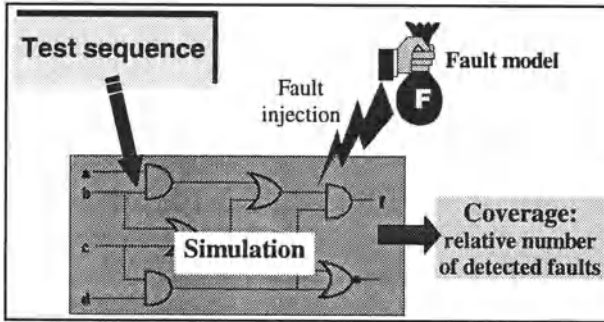


Figure 12.24. Fault grading by fault simulation

This approach is greatly used, as it is relatively simple to implement and can be adapted to numerous situations (system modeling and fault model). However, it involves a lot of processing and memory size; hence, it requires high performance computers and uses long run times. Indeed, it is necessary to apply the test sequence for each fault of the fault model injected in the system modeling. This leads to the simulation of millions of events. Numerous variants of fault simulation algorithms and implementations have been proposed in order to improve its performance (parallel processing, etc.).

In the next sub-sections, we analyze the fault grading methods based on fault simulation (sub-section 12.3.3.2), and we introduce the principal computing methods implementing fault simulation (sub-section 12.3.3.3).

12.3.3.2 Fault Grading by Fault Simulation

The methods presented in this section are used in today's IC Computer Aided Design tools. There are three main families of fault grading

approaches based on fault simulation techniques: 1) *probabilistic*, 2) *deterministic*, 3) and *statistical*. These three families are compared in Table 12.2 according to 4 criteria: *cost*, *speed*, *accuracy* and *diagnosis* efficiency. The assigned grades are justified afterwards.

Criteria	Probabilistic	Deterministic	Statistical
Cost	Low	High	Medium
Speed	Fast	Slow	Fast
Accuracy	Fair	Best	Good
Diagnosis	Good	Best	Poor

Table 12.2. Comparison of fault grading approaches

1. Probabilistic fault grading (PFG)

The *probabilistic fault grading* method provides an estimation of the fault coverage rather than an exact determination. It is a fast and relatively inexpensive method close to conventional logic simulation. The associated tools can be used on the same platform as the one used for design, which eliminates the needs for *netlists* and test vector conversions. The principle is based on an analysis of values which appear in different *nodes*, that is to say the interconnection lines between the system’s structural elements such as the gates, flip-flops, etc. The node activity is evaluated in terms of *controllability* and *observability*. The diagnosis provides a list of nodes which have a weak activity. The approach can be implemented as an interactive tool: the engineer uses this tool, searching to improve the testability of the less active nodes. However, since it makes no use of the strobe placements used by the physical tester which tests the real component, PFG may provide wrong information: a fault may be declared as observable when, in fact, it will not be observed due to a lack of strobing at the necessary time.

The analysis of the results provided by the PFG shows that the results are almost similar to the results obtained by more accurate methods. However, this small difference may be unacceptable for high fault coverage requirements.

2. Deterministic fault grading (DFG)

Deterministic fault grading is the most accurate of the three studied methods. It compares the simulation results of a faulty design (a copy of the design with a fault injected) with the outputs coming from the original design. If differences are found between the results of these simulations, the

injected fault is declared as detected. The ratio of detected faults to the total number of potential faults is used as a measure of coverage. This technique requires a lot of time and computing resources, as each possible stuck-at fault is injected into the design model and processed by a new simulation. Several different variants to DFG have been proposed to improve its efficiency. These methods include various simulation algorithms:

- grouping of equivalent faults, also known as *fault collapsing*,
- and making use of customized hardware platforms (*accelerators*).

Fault collapsing is a general technique used for reducing simulation time by identifying equivalent faults and simulating only one fault for each equivalence class. Before the fault-collapsing, it is often interesting to check if any of the nodes remains at either a '1' or a '0' during the test period. If a node always stays at '0', then a stuck-at '0' on the node cannot be detected. Similarly, a stuck-at '1' cannot be detected on a node that is always at a '1' level. No-activity faults and any collapsed faults that depend upon them are usually counted as non-detected. Checking for no-activity nodes can significantly reduce execution time during early stages of a test-pattern development where these kinds of faults are very likely.

In general, the implementations on general-purpose processors tend to run slower, but at a lower cost. *Test accelerators* are customized hardware which run much faster, but the hardware costs can be prohibitive.

Other factors can improve the accuracy of DFG implementations. These include propagation delay simulation, an increased number of simulation events, and actual strobe-placement information of the real test.

With respect to diagnosis information, DFG is efficient to providing the actual status (detected or not) for each potential fault in the design. Various reporting formats are typically provided for determining which parts of the design require the most improvement. In some implementations, information is also available which indicates effectiveness of each test vector to detect faults. Reordering the vectors and removing the unnecessary ones may optimize test patterns.

Fault dictionaries associate vectors (or sequences) with their detected faults. They may be generated from the results of DFG, helping in the diagnosis of probable causes during failure analysis of failed devices.

3. Statistical fault grading (SFG)

A strong reduction in the cost of DFG can be obtained by applying deterministic fault simulation to sub-sets of the potential faults of the given fault model. By choosing a random sample of these faults, *statistical fault grading* provides a close approximation of the DFG results, while requiring only a small fraction of the run time. The confidence interval of the results is

determined by the size of the taken sample.

In theory, SFG provides the speed and cost advantages of PFG, while providing accuracy offered by deterministic techniques. However, since only small portions of the potential faults are actually investigated using SFG, diagnosis information is very limited.

If the purpose in fault simulation is to simply obtain an accurate measure of the effectiveness of the test patterns, a DFG run might be appropriate. However, if the aim is to make use of the results to increase fault coverage to a certain target level, several iterations may be required. The low cost and fast run times of PFG are better suited to this type of activity. In most cases, cost and time are very important factors, but potentially inaccurate results cannot be tolerated. Therefore, a combination of both PFG and DFG is advised.

12.3.3.3 Fault Simulation Implementation

Fault simulation is the base to most fault grading methods. The implementation choices of the fault simulation have an impact on the performance of the fault grading tools. We will introduce the four main approaches: *serial*, *parallel*, *deductive*, and *concurrent* fault simulation.

1. Serial Fault Simulation

The less complex fault simulation algorithm is *serial fault simulation*. Essentially, it involves automating a series of logic fault simulations. Prior to performing them, it is necessary to first process a conventional logic simulation of the faultless system modeling by the test sequence inputs, to get and store the fault-free states of the test nodes at the strobe time-points. Then, faults are selected one at a time from the fault model. Each fault is applied to the circuit modeling and a logic simulation of the faulted circuit modeling is performed. The faulted test values are compared with the stored reference values. If a difference exists, the fault is flagged as detected. Then, the program proceeds to the next fault. If the faulted logic simulation reaches the last strobe, the fault is undetected.

The method has the advantage of easy implementation as hardware or software tools. It has the disadvantage of longer run times than most of the more sophisticated methods.

Several improvements that can be made to optimize serial fault simulation in term of speed. The first improvement deals with the optimization of the logic simulation algorithm and data structure (memory size, execution speed). Serial fault simulation can also be used as a first approach prior to development of a faster and more sophisticated method. Indeed, all of the front-end and back-end coding for fault-data input, fault

selection, fault injection and initialization could be developed and debugged for serial simulation. This serial simulation version can then be used as a prototype to verify the answers of more sophisticated versions that follows.

2. Parallel Fault Simulation

Parallel fault simulation takes advantage of the fact that each of the faulty circuits has the same topology as the reference circuit. Only the node's states are different. The basic idea is to group together the states of the reference circuit and several faulty circuits into a single word, a 'state word'. Then, the simulator treats globally these words in one pass instead of treating each case separately. This method was originally designed for simulators with simple unidirectional gate-level simulation. However, it can be extended to handle more sophisticated components.

Parallel fault simulation was used in some of the earlier generation of fault simulators (TEGAS, LASAR, HILO-2, FMRSIM, and EBNR programs). There are several limitations to this method. An increasing number of circuits running in parallel may reach the point from which the performance decreases. This classical phenomenon associated to parallel processing is due to two effects:

- All of the parallel faulty circuits have to be simulated until the last circuit of the group has its fault detected, or until the testing is completed. With serial simulation, analysis of each faulty circuit can be stopped as soon as its fault is detected.
- The more circuits are analyzed in parallel, the greater is the likelihood that an entire state word will have to be computed and updated for only one of the faulty circuits. In extreme cases, parallel simulation becomes like a group of serial fault simulations.

Ignoring the topological constraints that parallel fault simulation imposes, the industrial experiences shows that the method is still limited in speed to at most about 30 times faster than serial fault simulation. In fact, depending upon the circuit and the test pattern, it may be less than an order of magnitude faster than an optimized serial fault simulator.

3. Deductive Fault Simulation

Deductive fault simulation can simulate a relatively large number of faults in one pass. Instead of computing signal values for all considered faults, this method computes signal values for the fault-free circuit, and deduces the faults that will cause each signal to have a value different from the value of the fault-free circuit. This method is ideal for two-valued simulation (0/1), and it has been extended to handle the unknown value (0, 1

and x). For each analyzed element of the structure, the deductive simulation method determines the output fault-free values and local fault lists, from their input values and incoming fault lists. The fault-free values of an element output are computed whenever one or more of its input fault-free values change. The output fault list is computed whenever any of its input fault-free values or fault lists changes.

One problem involved is that some faults on variables which propagate back to the same input variables (loops), must be deleted from fault lists. If a fault-free signal is unknown, no information about faulty value of that signal is maintained in deductive simulation. Therefore some loss of information may occur.

4. Concurrent Fault Simulation

The basic idea of *concurrent fault simulation* is to simulate the reference circuit, while at the same time simulating many faulty circuits. The efficiency is improved by simulating only the section of each faulty circuit that differs from the reference circuit. Its high efficiency is due to the fact that, in most cases, the node states of each of the faulted circuits are nearly identical to the corresponding states in the reference circuit. Since each partial faulty-circuit simulation will be quite fast, this method will only be efficient if many faulty circuits are run concurrently with the reference circuit in order to share the overhead of the reference circuit simulation.

The concurrent fault simulation algorithm is potentially the most efficient and fastest method for fault simulation. Its main drawbacks are:

- the implementation is more complex than for the other methods,
- it may require much more computer memory to run efficiently, compared to other methods.

5. Comparison

The performance of fault simulation algorithms is important because it can consume many hours of CPU time for rather small circuits. A comparison of *parallel fault simulation* and *deductive fault simulation* shows that the second method is faster, but the first one is better for small, highly sequential circuits.

Another important point is that the *deductive fault simulation* is more pessimistic than *parallel* and *concurrent* fault simulations. *Parallel fault simulation* and *concurrent fault simulation* are equivalent in accuracy.

12.3.4 Test Pattern Generation of Combinational Systems

12.3.4.1 Optimal Test Sequence Generation

The obtaining of an *optimal test sequence* in terms of number of vectors (length) is a complex problem. To understand this problem, we can start with the analysis of simple n -input gates; it shows that $n+1$ test vectors are necessary. For example, the minimal test sequence of a 2-input AND gate is $\langle 01, 10, 11 \rangle$, and the minimal test sequence of a 2-input OR gate is $\langle 00, 01, 10 \rangle$. Example 12.4 performs such analysis for a 3-input NAND gate. From these basic experiments, it is generally not difficult to derive minimal test sequences of small circuits, such as in Example 12.5.

In a more general manner, the research of a minimal cover in a cover table calls for operational research methods such as *Branch and Bound* methods. These methods are only of interest for understanding test problems. Indeed, they are very complex and the determination of a cover table is intractable for industrial circuits. Hence, we will propose a heuristic approach to test pattern generation in next sub-section.

Example 12.4. Optimal test sequence of a NAND gate

We consider again a 3-input NAND gate test with a single stuck-at fault model (see Example 12.1 in sub-section 12.2.3). We had proposed a minimal test sequence: $\langle 011, 101, 110, 111 \rangle$. How can this sequence be obtained? The goal is to find a minimal set of input vectors detecting all the faults. For this, we start with the *cover table* (see Table 12.3) already obtained.

Input vectors	Test coverage			
	a	b	c	d
0 0 0	-	-	-	0
0 0 1	-	-	-	0
0 1 0	-	-	-	0
0 1 1	1	-	-	0
1 0 0	-	-	-	0
1 0 1	-	1	-	0
1 1 0	-	-	1	0
1 1 1	0	0	0	1

Table 12.3. Coverage table

A test sequence with 100% fault coverage must provide a ‘0’ and a ‘1’ at least once for each column in this table. We see that, in order to test the

stuck-at '0' and the stuck-at '1' of input a , it is necessary to take the test vectors 011 and 111. We can easily see on the table that these two test vectors detect all faults except, the stuck-at '1' of b and c input. The test of these two remaining faults requires taking vectors 101 and 110. Therefore, there is only one optimal test sequence.

Note. The minimal Toggle Test sequence for this NAND gate is $\langle 000, 111 \rangle$. This sequence obviously does not test all the considered faults: it only tests 5 of them: a^0, b^0, c^0, d^0 , and d^1 (a^0 represents the stuck-at 0 of line a).

Example 12.5. Optimal test of a small circuit

Let us consider the simple AND-OR circuit of *Figure 12.25*. In order to detect all stuck-at faults of this circuit, each AND gate must receive at least the input vectors 01, 10 and 11, and the OR gate must receive the input vectors 00, 01 and 10. Additional constraints must be added to propagate the errors to the output f . For example, we must not apply 11 simultaneously to both AND gates. Exercise 12.6 invites you to make this study. The optimal test sequence has 4 test vectors: $TS = \langle 110, 011, 010, 101 \rangle$.

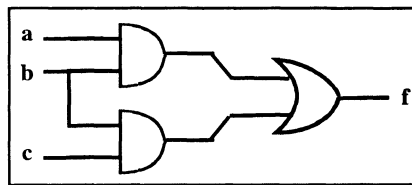


Figure 12.25. AND-OR circuit

12.3.4.2 Heuristic Test Pattern Generation

Typically, the obtaining (automatic or semi-automatic) of a test sequence which ensures a given coverage rate (for example 80%) combines two types of methods which were introduced in the previous sections: the test generation and the coverage evaluation.

Indeed, a test vector obtained for the detection of a given fault also detects a set of other faults. We look for this set of faults by a coverage assessment of the test vector chosen to detect the first fault. Then, we delete all these faults before restarting the search for another test vector to detect a fault not yet revealed. This process is repeated as long as necessary, that is to say until the fixed fault coverage objective has been reached. This procedure was introduced and used by IBM. It was then used and improved by numerous other authors and companies.

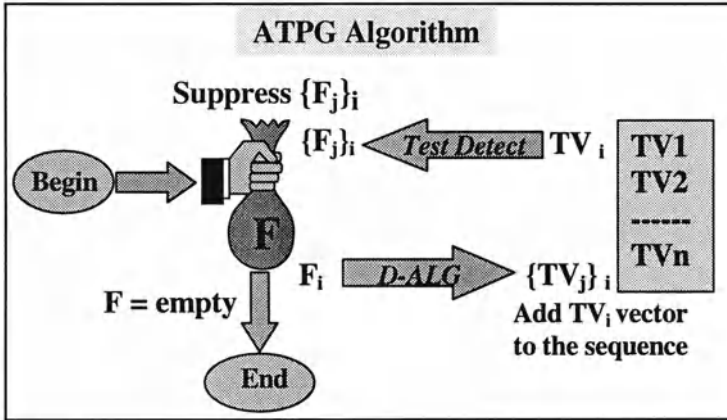


Figure 12.26. Test sequence generation

Figure 12.26 illustrates this algorithm with the alternative use of two tools from IBM: D-ALG for the determination of a test vector which detects a given fault, and TEST DETECT to look for faults covered by a given vector. At the beginning of the algorithm, the set of faults to test, F , is made up of all the faults of the fault model, and the test sequence is empty. We choose a fault F_i and, by using D-ALG, we look for an input vector TV_i which detects it. This is the first vector of the test sequence. Then, this test vector is applied to TEST DETECT to determine all detected faults. All these faults are removed from the set of faults F . This process is re-iterated with another fault to test, until the set of faults to be tested is empty.

With this method, the optimality of the obtained test sequence cannot be guaranteed. Heuristics allow faults to be chosen in a way that accelerates the test generation process and obtains shorter test sequences.

This iterative process often starts with a functional test sequence coming from the product’s design stages (Chapter 10). The electronic circuit specialists affirm that these functional sequences typically provide coverage rate from 50 to 70 percent.

12.3.5 Test of Sequential Systems

12.3.5.1 General Problem

In reality, very few logical circuits are totally combinational: this is the case of certain calculation circuits, coding/decoding systems, or code transformers. On the contrary, the majority of circuits are sequential, that is to say the outputs depend on the inputs and also on the *internal state* (notions of memory or time). Consequently, it is important to question about the adaptation of methods previously exposed, or to imagine specific

methods intended to sequential systems. Despite the numerous studies which have been carried out, none of them has led to efficient or accessible methods. We can quote the example of the method based on *formal identification* of an automaton (system behavior) to provide a test sequence at system level. This approach which makes not precise hypotheses on fault model was first studied in the 1960s for electronic systems. It is still studied at system level, for the *conformance testing* of communication protocols.

Why is the sequential system test so difficult?

On the one hand, the research of test sequences becomes very complex in the case of *synchronous circuits* (special variables called clocks control the time evolution of the circuits), and terribly much more in the case of *asynchronous circuits* (the circuits reacts asynchronously to events applied to the primary inputs). Unfortunately, some faults can increase the number of internal states, transforming a sequential synchronous system into an asynchronous system, and even provoking oscillatory behavior.

On the other hand, these sequential system's controllability and observability problems often lead to test sequences which have a prohibitive number of vectors! Indeed, a small changing in the event occurrence date may disturb the behavior. Consequently, the test sequence must take this large number of situation into account.

Numerous methods have been proposed to test complex sequential circuits such as micro-controllers or microprocessors. They use functional approaches based on high level modeling tools, such as HDL. Functional fault models are implicitly or explicitly attached to these models. A classical example is the STG (State Transformation Graph), which can be deduced from a HDL description of a system. A path sensitizing method is then applied to this graph to determine a test sequence.

The test of complex circuits, whether combinational or sequential, generally find empirical solutions based on functional sequences which are then improved using processes using together fault simulation and the manual research of new test vectors. The only satisfactory solutions to this problem use BIT type techniques presented in Chapter 14.

Example 12.6 proposes a study of a simple synchronous circuit which links fault generation at state graph level and fault coverage at gate level.

In the next sub-section, we will consider the very special case of RAM.

Example 12.6. A MOORE synchronous sequential circuit

Figure 12.27 shows an example of a Moore type sequential synchronous circuit whose outputs only depend on the circuit's internal state. This circuit has one input x and one output z . It is described at a behavioral level and at a logical level. The behavioral model is a 4-state automaton. This model has been implemented as a logical circuit with 1 INVERTER, 5 NAND gates

and two D Flip-Flops. The coding of the 4 states is shown in the figure.

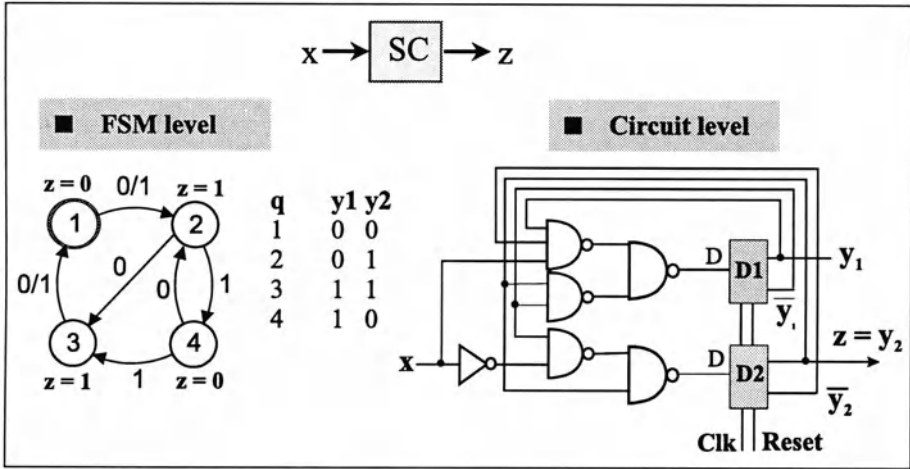


Figure 12.27. Test of sequential circuits

In Chapter 5 (Exercise 5.3), we have analyzed this circuit and determined some failures associated with two faults altering the logical structure.

We consider now two *functional test sequences* associated with the automaton level which are defined from the initial state ‘1’:

- the first sequence of length 4 goes through all the graph’s states: $TS1 = \langle 0\ 1\ 1\ 0 \rangle$,
- the second sequence of length 9 goes through all the graph’s arcs: $TS2 = \langle 0\ 1\ 1\ 0\ 1\ 1\ 0\ 0\ 1 \rangle$.

A fault simulation at logical circuit level has been carried out with the help of the *VeriFault* program of the CAD *Cadence* industrial tool. It gave the respective coverage rate: 65% for the first sequence and 92% for the second one. We notice that a ‘functional’ type test sequence established at the automaton level does not test all the single stuck-at 0/1 faults. The structural approach at the logical or electronic level is necessary to complete the test and guarantee 100% coverage. Exercise 12.7 comes back to the test of this sequential system at the state graph level.

Note. A 100% fault coverage rate is not a certainty of the total absence of creation faults and breakdown faults due to ageing. The meaning of this rate is relative to the system model and the fault model being considered. We detect all the considered situations which transform these faults into errors in this system model.

12.3.5.2 Test of Random Access Memory

A Random Access Memory (RAM) is a special sequential circuit which stores words of fixed length. It has two access modes: *write* to enter one word at a given address, and *read* to extract one word from a given address. Logically speaking, a RAM can be represented as a collection of registers (first diagram of *Figure 12.28*).

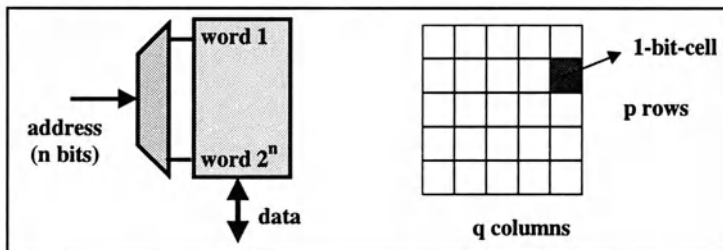


Figure 12.28. RAM representation

So, a first simple functional test sequence can easily be imagined: a series of known words are sequentially written, and then sequentially read and compared to the original words; then the same test is applied with the complementary pattern. This 'linear' test sequence of length $N = 2^{n+1}$ is unfortunately not excellent to detect real internal faults. Indeed, a physical memory does not conform to the previous model. To minimize the silicon surface, it is organized as a square matrix (p rows \times q columns) of elementary bit-cells (second diagram of *Figure 12.28*).

The physical study of real circuits has revealed different classes of possible hard and soft faults creating different classes of errors: one cell, one row, one column, address errors, but also mutual influence between cells and, even worse, data sensitive errors which depend on the bits 0/1 which are stored in the neighborhood of a given cell.

Several methods have been proposed to detect such fault classes. Their complexity in number of test operations (read/write) is generally relative to their coverage. Let us just mention some of these methods which take the electronic technology into account to be closer to the real faults:

- *checkerboard 0/1*: the matrix is written with a checkerboard pattern, read, and the operation is repeated with the complementary pattern;
- *marching*: the matrix is initialized with a pattern (e.g. 0's), then each bit is successively read, complemented, written back, and read again, in increasing order of the bits, and finally in decreasing order;
- *walking columns*: the matrix is initialized with a pattern (e.g. 0's), then the first column is written to the complementary values, the matrix is

read, and this process is repeated by shifting the column pattern to the right; this technique can also be applied to a diagonal whose pattern is shifted to the right in the matrix;

- *galloping* or *ping-pong*: the cells are partitioned into several groups called contamination group; for example, if the contamination group of a cell is the column, after initializing the matrix, each cell is successively complemented and, each time, all the cells of its column are read.

12.4 EXERCISES

Exercise 12.1. Signature testing

A component tester applies a sequence of 1024 input 10-bit vectors to a device under test (DUT) having 10 inputs and one output. The circuit answers to this input sequence with an output sequence of 1024 bits. A sequential compaction treatment is applied to this output sequence in order to produce a 16-bit signature. So, the 1024-bit stream is split into 64 16-bit words noted A_i . These successive words A_i are 'accumulated' in a 16-bit register R by an XOR logical vector operation: $R = R \oplus A_i$.

1. Which classes of failures cannot be observed with this technique?
2. From this, can we deduce the electronic component's fault classes which have not been tested by the tester?

Exercise 12.2. Toggle test sequence

Consider an adder whose logical gate schema is given in *Figure 12.29*.

Find the shortest possible input sequence which sets each gate input/output line to '0' and to '1'.

We will refer to this exercise in the next chapter, in order to evaluate the efficiency of this *toggle sequence*.

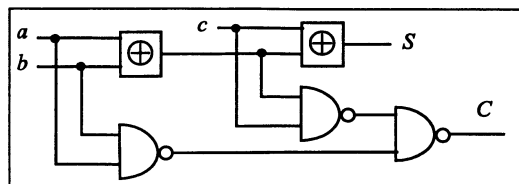


Figure 12.29. Full adder

Exercise 12.3. Test of components

Imagine a component production chain whose *yield* is $y = 90\%$. We suppose that we apply a test sequence to each produced component (test of

100% of the components). This sequence has n test vectors, each one having a duration of 1 time unit. The sequence has a coverage rate of $c = 80\%$ of faults of the fault model considered. We also assume that the defective components are detected on average after $n/2$ test vectors.

1. Determine and analyze the expression of the average test length.
2. Calculate the rate of defective produced components which are declared as 'good' by the test.
3. We now suppose that only $t = 70\%$ of the produced components are tested. Refer to the previous two questions.

Exercise 12.4. Fault coverage

We consider a NOR gate with 3 inputs, a , b and c and one output d . Analyze this gate's functioning when affected by single stuck-at 0/1 faults of the a , b , c and d signals.

1. Find the test sequence having the shortest length (this sequence will be called *optimal* test sequence).
2. Trace the coverage curves of i) the exhaustive test sequence, ii) the optimal test sequence, iii) a toggle test sequence.

Exercise 12.5. Simple fault diagnosis

Continue the fault analysis of the NOR gate in the previous exercise in order to distinguish the different faults.

Are certain faults non-distinguishable, that is *system equivalent*?

Exercise 12.6. Optimal test sequence

Analyze the circuit of Example 12.5 to find the optimal test sequence(s), according to a single stuck-at fault model. Justify any test vector choice.

Exercise 12.7. Sequential circuit testing

We consider the synchronous sequential system in *Figure 12.27*. We suppose that the initial state is state 1.

1. Check that the input sequences proposed in paragraph 12.3.5, *ST1* and *ST2*, respectively explore all the states and all the arcs of the graph. For each sequence, determine the sequence of the states and the outputs produced for each sequence.
2. Follow the logical evolution of the signals on the logical circuit for the same test sequences. Is each line of the circuit set to '0' and to '1' (consequently is it a toggle test)?
3. Comment on the problem experienced in detecting all the circuit single stuck-at faults.

4. Discuss the hypothesis concerning the initial state (state 1) of this sequential system. How can we guarantee that the system is really in state '1' at the beginning of the test?

Chapter 13

Structural Testing Methods

In the previous chapter we carried out a general presentation of the various methods, which allow the suppression of technological faults during the manufacturing and the operation stages of the lifecycle of the product. We will now magnify some of the technical aspects of structural testing, focusing on hardware and software technologies.

In a first part (sections 13.1 to 13.5), we analyze some simple methods that are based on fault models. Then, we consider the techniques of structural testing that do not explicitly make reference to fault models (sections 13.6 and 13.7). Finally, section 13.8 presents mutation testing which covers both aspects, either with or without fault model.

13.1 GENERATION OF LOGICAL TEST BY A GATE LEVEL STRUCTURAL APPROACH

The functional test aims at exercising all possible behaviors of a system to detect the presence of faults by observation of the outputs. Hence, all faults activated by these behaviors are detected. Structural testing takes the structure of the system into account by analyzing all possible behaviors of its elementary components. We have already developed the interest of structural testing. We have also highlighted the precautions that must be taken concerning the interpretation of the *coverage rate* which provides a measure for the efficiency of test sequences. In this chapter, we go deeper into the structural testing methods for hardware and software technology.

In accordance with the technology, these techniques are initially distinguished by the way that we observe the behavior of the components within the structure.

Concerning software technology, structural testing attempts to activate every possible component of the structure. This test is not a priori based on precise hypotheses about the faults or the errors that might affect the program. The activation is characterized by a property on the functioning of the components. Let us consider the simplest example of activation: “a statement has been executed”. The considered component is the statement; the property on its functioning is the fact of being executed. The corresponding coverage rate will measure the efficiency of the sequence, which is equal to the number of components for which the property is true divided by the total number of components. The number of statements executed divided by the total number of instructions in a program is an example of coverage rate.

On the other hand, the structural testing of electronic systems is mainly based on fault or error models. A good test sequence should be capable of activating the faults in the model that are present in the system, and of propagating an *immediate or primitive error* induced by this fault or associated to the error model. For example, a structural test sequence for a circuit modeled at gate level will have to detect the stuck-at 0/1 errors (error model) of each of the constituent gates.

The second approach is much more precise than the first one, but it is based on strong hypotheses that must be valid.

- First of all, it supposes that *faults or errors do not affect the structure of the system, or modify it only slightly*. For example, a connection between two components *C1* and *C2* cannot be suppressed and replaced by a connection between *C1* and a component *C3*.
- Secondly, it supposes the knowledge of a *realistic error model*.

In the case of logical level models of electronic systems, these two hypotheses are acceptable. First of all, the chosen faults do not affect the structure. They uniquely concern the functioning of the components, such as the logic gates. Furthermore, the misfunctionings are known and can be characterized by an error model (such as the stuck-at errors). Finally, experience has shown the relevance of these two hypotheses for manufacturing and operation phases. Unfortunately, these two cases are not true for software models or abstract models of electronic systems (HDL, Petri nets, etc.). For software technology, design faults modify the structure of the system in an unpredictable manner (there is no fault/error model).

Certain techniques attempt to benefit from the two points of view. The studies concerning *mutation testing* of software draw their inspiration from the approach adopted in electronics. For example, we measure the efficiency of a test sequence by evaluating its capacity to detect the replacement of a ‘+’ operator by a ‘-’ operator in an arithmetic expression. Thus, we do not

change the structure of the program but we conceive a known modification of the behavior of an operator.

First of all we are going to study the techniques of structural testing that are based on error models. They will be illustrated by means of an error model of 'stuck-at' type that affects a structural model of a circuit at gate level. We are successively going to: determine the test sequences that allow us to uncover the presence of errors of this model that corrupt the system (section 13.2), evaluate the errors of a model that are detected in the system by a given sequence (section 13.3), define a diagnosis sequence that allows us to localize a particular error, which is a part of the model and affects the system (section 13.4), then study the influence of passive redundancy on the detection and the diagnosis of these errors (section 13.5).

The next two sections (sections 13.6 and 13.7) assume no fault/error models. The principles of the techniques presented will be illustrated by software applications. Section 13.6 tackles the structural test without fault or error models, and section 13.7 introduces the diagnosis methods without fault or error models.

Finally, we present the technique of mutation testing in section 13.8. We thus show how the techniques that are based on error models have influenced the testing of software applications.

13.2 TEST GENERATION FOR A GIVEN ERROR

In this section we present a test generation technique, which allows the detection of faults/errors that belong to a given model. We refine this presentation by considering a system modeled by logical gates (NOT, AND, OR, NAND, NOR, XOR) and affected by single stuck-at 0/1 faults.

13.2.1 Principles of the Method

An input vector applied to a circuit detects a fault that affects this circuit if it satisfies three conditions: First, this input must induce an active signal that *reaches* the component affected by the fault. Then, this signal must *activate* the fault by transforming it into a local *error*. Finally, this error must be *propagated* in the structure of the system, until it reaches at least an output where it will be observable by the external tester. In this way, we want to artificially provoke the activation of a fault as an observable failure.

We will analyze a simple and intuitive procedure, called *path sensitizing*, for the search of the test vectors that detect a given fault f belonging to the fault model. This procedure, illustrated by *Figure 13.1*, has four stages:

1. *activation* of the fault as a *primitive error* ($f \rightarrow e$),

2. *backward propagation* or *tracing*, towards the primary inputs, in order to find the input vectors producing the preceding activation,
3. *forward propagation* along a chosen path, in order to allow the observation of the primitive error at the outputs of the circuit,
4. *justification* or *consistency* operation which verifies that the local actions implied by the preceding steps are coherent in the whole circuit, and finds the input vectors that satisfy them.

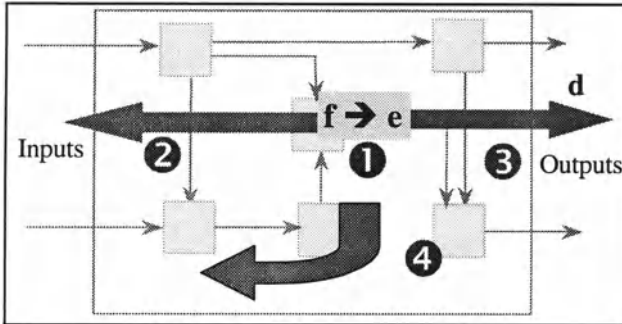


Figure 13.1. Procedure for testing a fault

The actual implementation of this procedure is explained below with the help of an example. It can lead to the exploration of a decision tree where several choices are possible: the choice between several error propagation paths, or the choice between several values of certain variables. When these choices exist, the previous four-staged procedure must be re-iterated. If no solutions are found at the end of the procedure, it means that the fault is *undetectable*.

13.2.2 Activation and Backward Propagation

First of all, we have to activate the fault f into an error e and find the input vectors that allow this activation. This stage is called *fault activation*. When the fault is a single stuck-at 1 (respectively 0) of a line, we must set this line to 0 (respectively a 1). Hence, the fault is activated as an error. This means that without fault the line takes the value 0 (respectively 1), and when the fault is present this line takes the value 1 (respectively 0). According to the case, we can either retain the symbolic notation e , or specify the fault type by writing: $1 \rightarrow 0$ or $0 \rightarrow 1$. This error e is called *primitive* or *immediate error*.

Next, we perform a *backward propagation* (or *backward tracing*) to attempt to propagate this knowledge back to the primary inputs. This

knowledge is defined as *constraints*, specifying expectations on line values to activate the fault.

Example 13.1. Fault activation and backward propagation

Consider the small circuit in *Figure 13.2*, and the fault f , stuck-at 1 of line d . To *activate* f , we must place a 0 value on this line, which is the output of the AND gate. This constraint is now propagated *backwards*, and we find three possible values to put on a and b : 00, 10 or 01. At the end of these two first stages, the fault has been transformed into an error on line d : an expected value 0 which becomes a 1 when the fault is present in the circuit. We see that the input c still has no constraint.

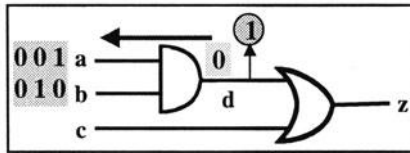


Figure 13.2. Fault activation

13.2.3 Forward Propagation

Now, the primitive error must be propagated to the output so that it is observable by the external tester. The *forward propagation* stage consists in expressing the local constraints which allow the propagation of the primitive error, from gate to gate, along one or several paths. When a *propagated error*, noted ei , reaches one input of a given gate, we have to do one of the following, according to the situation:

- if we want that this error propagates through the gate, we must find appropriate values to apply to the other inputs so that ei passes and gives another error ej ,
- if we do not want that this error propagates through the gate, we must find values to apply to the other inputs in order to maintain the output of the gate at its fault-free value.

The second case is desired if we want to control the error propagation along a predefined path.

This error propagation mechanism is analog to the propagation of electric signals (raising or falling edges or pulses) through the logic circuit. *Figure 13.3* shows examples of error propagation constraints through logic gates.

According to the type of gate that is crossed, the output error is 'in phase' (noted e) or 'in phase opposition' (noted e') with the input error (noted e). The first case corresponds to the crossing of an AND or an OR gate, whereas

the second case corresponds to the crossing of an inverter gate such as the NAND or the NOR gate. For example, to allow an error to cross an AND gate, all other inputs must be set to value '1' (which is the neutral element of AND); therefore, the error passes through the gate and gives an error having the same characteristic (it is called 'in phase'). On the other hand, the XOR gate always allows an input error to pass through, the output error being of type e or e' , according to whether the second input has the value '0' (neutral element of XOR) or '1'.

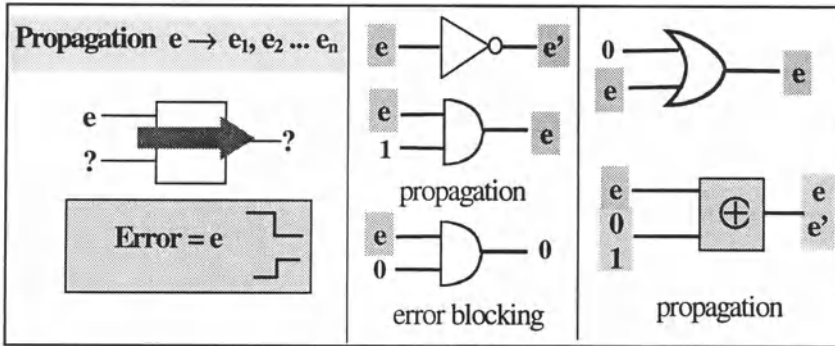


Figure 13.3. Error propagation constraints

In a very general way, the errors can propagate themselves along several paths. The 'mother' error (*primitive*), which is produced by the first activation of the fault, thus gives birth to several 'daughter' errors that propagate themselves along separate paths. When they re-converge on the same gate (*reconvergent fan-out structure*), we have to examine *multiple* error propagation across the reconvergent gate. Hence, a symmetrical gate (AND, OR, NAND, NOR) propagates two errors 'in phase', and blocks two errors in 'phase opposition'. For example, let us consider an AND gate that receives two errors in 'phase opposition', $1 \rightarrow 0$ and $0 \rightarrow 1$. In the absence of a fault, the output has the value 0 (1 AND 0), and when the fault that provoked these errors arises, we also have 0 at output (0 AND 1). Therefore, this error is not observed at output.

Example 13.2. Error propagation

Let us consider again the example of the stuck-at 1 fault at line d of the small circuit in *Figure 13.2*.

To reach the output, the primitive error e ($0 \rightarrow 1$) must pass through the OR gate. We must therefore place a value '0' at the input of this gate, i.e. the primary input c . Hence, we obtain three test vectors: $(abc) = (000, 010, 100)$. The induced failure is a $0 \rightarrow 1$ ($z = 0$ if the fault is not present).

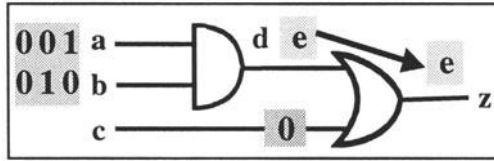


Figure 13.4. Error propagation

13.2.4 Justification

The three previous steps have led to the establishment of constraints on the values of certain primary inputs and/or internal lines of the circuit. The *consistency* or *justification operation* checks that these local constraints are compatible and find the value of the test vectors, if they exist.

Example 13.3. Justification

To illustrate the problem of compatibility or non-compatibility of local constraints, let us analyze the small circuit in *Figure 13.5*. We assume that an error e occurs at line a . The propagation of this error at output f brings the local constraints '1' and '0' noted in the figure. Nevertheless, these two constraints are incompatible, since the OR gate (noted A) cannot have an input value 1 and an output value 0 at the same time. Hence the justification procedure reveals an inconsistency. To conclude, no test vector may propagate the error as a failure. This error e is not detectable, because it corresponds to a passive redundancy.

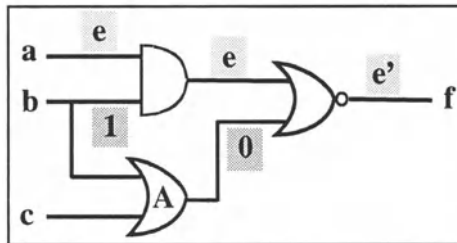


Figure 13.5. Consistency

13.2.5 Complete Study of a Small Circuit

Example 13.4

Now we are going to illustrate the previous four steps with a complete example. Consider the circuit in *Figure 13.6*, and the stuck-at 0 fault of line III, noted α in the figure.

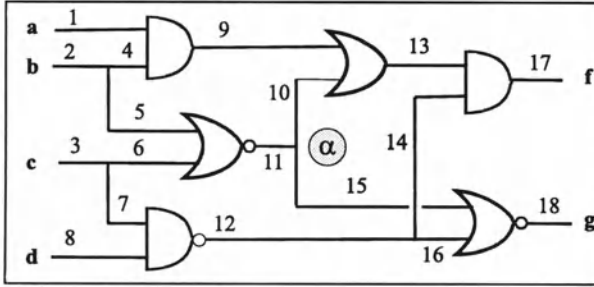


Figure 13.6. Gate structure

13.2.5.1 First Step: Activation of the Fault

To activate the fault as a primitive error, the test vector must produce a value '1' at line 111 without fault. Thus, the occurrence of fault α will provoke an error noted e at line 111: '1' without failure, which becomes '0' when fault α occurs.

13.2.5.2 Second Step: Backward Propagation

Now let us see how to satisfy this constraint from the primary inputs. We go back (by *backward propagation*) from line 111:

$111 = 1 \rightarrow 15 = 0$ and $16 = 0$, which implies $b = 12 = 0$ and $c = 13 = 0$.

The situation after execution of these two stages is shown by Figure 13.7.

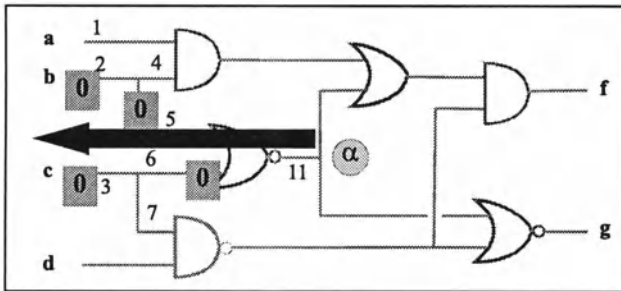


Figure 13.7. Steps 1 and 2

13.2.5.3 Third & Fourth Steps: Forward Propagation and Consistency

Let us come back to error e . Two paths, $P1$ (lines 11 - 10 - 13 - 17 = f) and $P2$, (lines 11- 15 - 18 = g) are candidates to propagate it, respectively on f and on g (Figure 13.8). We must successively try to propagate the primitive error through each path, then the two simultaneously, since we wish to find all the test vectors detecting this fault. If our objective is to find one test vector only, the procedure is stopped once a test vector has been found.

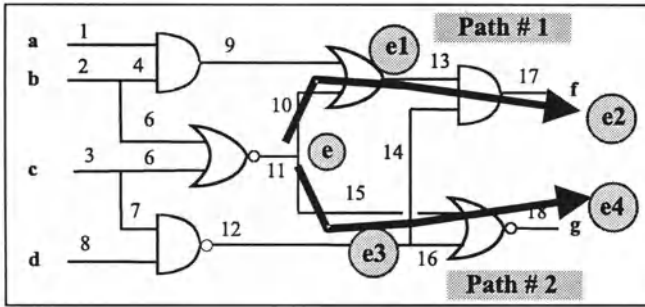


Figure 13.8. Two propagation paths

Path P1 only

The error e must first of all cross an OR gate: e is observable as an error $e1$ at the output of this gate if line $l9$ has the value '0'. If it had the value '1' then the output of the OR gate would be '1', and the propagation of error e would be halted! An error $e1$ is therefore produced at $l13$ at the input of an AND gate. To cross this gate, and finally reach the output, line $l14$ must be set to '1'. This last value blocks any propagation along path $P2$. Let us sum up the propagation constraints, which are called *path sensitizing*: $l9 = 0$, $l14 = 1$. These constraints must now be propagated backwards to the primary inputs to complete the test vectors and see if some inconsistency situations occur (for example the same line simultaneously taking values '0' and '1'):

- $l9 = 0$ implies $l1 = 0$ OR $l2 = 0$, which is satisfied since $l2 = b = 0$;
- $l14 = 1$ implies $l12 = 1$, requiring $l7 = 0$ OR $l8 = 0$, which is satisfied, as $c = l3 = 0$, thus $l7 = 0$.

Conclusion: the path $P1$ can propagate the error at output f for four test vectors noted symbolically: $(a \ b \ c \ d) = (- \ 0 \ 0 \ -)$. The fault provokes a '0' instead of '1' on f (failure).

Path P2 only

Error e has to cross a NOR gate giving g . This propagation requires that line $l16$ be equal to '0'. If not the output 18 would be forced to '0' and the error would not pass to g ! Let us see if the constraint $l16 = 0$ can be satisfied: by going backwards in the circuit, this implies $l7 = 1$ and $l8 = 1$; however, $l7$ has the value '0' since $l3$ has the value '0', thus it is impossible. Thus, no test vector allows detecting the fault at output g .

Path P1 and path P2

It is impossible to propagate the original error simultaneously through these two paths. Actually, we have just seen that the propagation condition

on f is that the line $l12$ is set to '1', and the propagation condition on g is that the line $l2$ is set to '0'! Therefore, the error is not observable simultaneously at outputs f and g .

To sum up, the fault α is detectable on output f by the four test vectors $(a\ b\ c\ d) = (-\ 0\ 0\ -)$.

13.2.5.4 Notes

The previous analysis has revealed a single decision level tree, which has three branches corresponding to the choice of two paths that are able to propagate the errors: $P1$ alone, $P2$ alone, and $P1+P2$. As a general rule, other decision nodes can be induced by backward and forward propagation mechanisms. Thus, Exercise 13.2 proposes the analysis of a fault that is more difficult to test, because it implies several choices, and thus lead to the exploration of a decision tree of several levels.

The reader might have noticed that the followed procedure is not very efficient to treat this fault. Indeed, at the end of step 2 (see *Figure 13.7*), we could have exploited the obtained values by a backward propagation: a logic '0' on line $l2$ implies a logic '0' on line $l9$; a logic '0' on line $l7$ implies a '1' on lines $l14$ and $l16$, which forces output g to logic '0'. Thus, the 4 test vectors are very easy to deduce. This remark shows that the simple path sensitizing method presented can be improved. For example, the path tracing based PODEM method uses an implicit vector enumeration by an orderly search algorithm. Exercise 13.3, proposes to analyze and improve a strategy.

13.2.6 Test of Structured Circuits

Still restricting our study to the case of combinational circuits, we will now briefly examine the case of *structured circuits*. Let us consider a simple 2-module structured circuit represented in *Figure 13.9*.

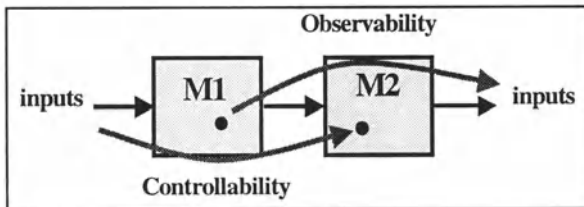


Figure 13.9. Structured circuit

Each module is supposed here to be individually completely testable (completely controllable and observable); the question is now to test these

modules from the primary inputs and outputs of the whole system. Module $M1$ is fully controllable from the inputs, but not necessarily fully observable from the primary outputs. Conversely, module $M2$ is fully observable from the primary outputs, but not necessarily controllable from the primary inputs. Therefore, defining a test sequence that will test the complete circuit is not a trivial issue. This testing problem is illustrated in Exercise 13.9 with a circuit made up of two full-adders in cascade to form a 2-bit number adder.

13.3 DETERMINATION OF THE FAULTS/ERRORS DETECTED BY A GIVEN TEST VECTOR

We are still considering a structured system and an error model, and we will illustrate the proposed method on a structure of logic gates and a stuck-at 0/1 error model. We assume that we are given an input vector from which we want to know the errors that it detects. Applied to a complete test sequence, this technique allows the evaluation of its coverage rate: *fault grading by structural approach*.

13.3.1 Principles of the Method

We are going to study a simple and intuitive method based on a structural analysis of the circuit using a propagation technique conducted in two steps (Figure 13.10): *forward simulation*, and *backward fault analysis*.

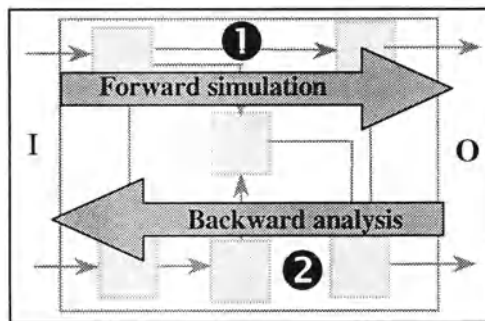


Figure 13.10. Determination of the coverage of a test vector

1. We establish the values of all the lines of the circuit structure by a *forward simulation*.
2. We go backwards in the circuit by means of a *backward fault analysis* along the paths or the logical layers, and each time we establish the faults that are detected.

A conventional simulation implements the first step. The second stage of the method consists in going back from the primary outputs to the primary inputs across the logic gate structure, in order to identify the faults that are detected and those that are not detected.

Figure 13.11 shows some examples of ‘backward’ analysis of logic gates. Output line c of the analyzed gate is supposed to be observed by the tester, and we are looking for the faults occurring in the circuit or the input errors that are detectable on c . Thus, in the case of an AND gate having (0 1) as input, we detect the stuck-at 1 of c , the stuck-at 1 of a , and nothing on the input b (this ‘non detection’ is noted by the symbol ‘-’). For the considered input configuration (0 1), we can say that, from an ‘observational’ point of view, the stuck-at 1 fault of a is ‘equivalent’ to the stuck-at 1 fault of c .

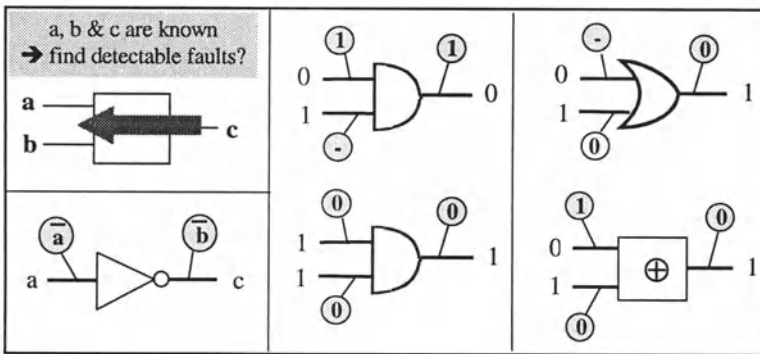


Figure 13.11. Backward analysis

The general principle of this method is therefore to exploit the observational fault equivalence by backward analysis of the logic structure. When a line cannot propagate any error, we also cannot detect any fault on all the components situated before this line. Indeed, a fault that is situated before this line will have to be activated by an error, and then this error must be transmitted up to the line in question. Now, by hypothesis, this error cannot be propagated further. This property allows the simplification of the analysis process, as we will see in the example that is treated later on.

This method is applied quite easily if the circuit does not have any *reconvergent fan-out structure*, that is to say if it does not have any signal that diverges on different paths before returning to a same gate. If such a case occurs, this procedure poses some problems, and a particular treatment must be carried out in order to detect any faults that are situated before the divergence points. Figure 13.12 illustrates this problem. A stuck-at 0 fault on line d provokes an error noted e_1 which is propagated along the two paths and arrives at lines a and b (errors e_2 and e_3). These two errors will be neutralized by the AND gate, which provides output $c = 0$ with or without

the fault. Thus, this fault of line *d* cannot be detected, whereas ‘backward’ analysis by observational equivalence would discover that this fault is detectable, since the fault of line *a* is detectable. In such cases, we have to consider whether the failures situated backward from the divergences can produce multiple error signals. That case will therefore be treated separately.

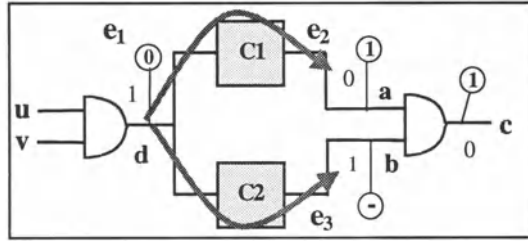


Figure 13.12. Reconvergent fan-out

13.3.2 Study of a Small Circuit

Example 13.5

Let us consider again the circuit of Example 13.4 represented in Figure 13.13, and let us suppose that we apply the input vector (1 0 0 0).

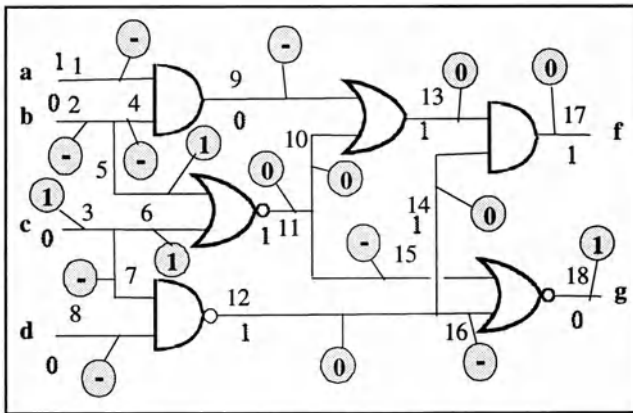


Figure 13.13. Fault coverage example

First of all, we note that this circuit has 3 cases of reconvergent fan-out:

- the signal on line *l2* takes 2 paths before reconverging on the OR gate giving the signal on *l13*: (*l4* - *l9*) and (*l5* - *l11* - *l10*),
- the signal on line *l3* takes 2 paths before reconverging on the AND gate producing the output *f*: (*l6* - *l11* - *l10* - *l13*) and (*l7* - *l12* - *l14*),

- the signal on line $l3$ takes 2 paths before reconverging on the NOR gate producing the output g : ($l6 - l11 - l15$) and ($l7 - l12 - l16$).

In *Figure 13.13* we have represented on each line the value obtained by normal forward propagation (step 1). From the values obtained by this first step, we go back from the outputs f and g , and we indicate the faults that are detected or are not detected in the circles (step 2). In this way, we detect the stuck-at 0 faults of lines $l17$, $l14$, $l13$, $l12$ and $l11$, and the stuck-at 1 faults of lines $l18$, $l6$, $l5$ and $l3$. We detect nothing on the other lines (marked -). The lines located before the divergence points are studied separately. The stuck-at 1 fault of line $l3$ propagates itself on a single path (which leads to f): it is therefore detectable. On the other hand, the stuck-at 1 fault of line $l2$ will create 2 errors at $l4$ and at $l5$. These two errors propagate themselves and arrive in phase opposition on the OR gate creating the signal on $l13$: they therefore neutralize each other and this failure is not detected on output g .

13.4 DIAGNOSIS OF A TEST SEQUENCE

Without going into details about the diagnosis test, we are simply going to present the principle of a technique and analyze the distinction capability of a test sequence applied to a small circuit. This will allow us to explain the role of a *diagnosis tree*.

13.4.1 General Problem of the Diagnosis

The problem of a diagnosis test has already been introduced in Chapter 12 (in section 12.2): from the observation of the responses of the product to the applied test vectors, how can we deduce the fault/error that is present?

Figure 13.14 provides some simple diagnosis examples allowing some faults to be distinguished. In order to distinguish the stuck-at 1 faults α and β of the first gate (AND), we look for a sequence that firstly detects the presence of one of these two faults (test vector 10, noted $TV1 = 10$ on the figure), and then separates them (second vector $TV2 = 00$). Hence, the resulting *distinguishing sequence* is $DS = \langle 10, 00 \rangle$. The reasoning is similar for the OR gate. By performing this analysis for several fault pairs, we realize that some faults cannot be distinguished. That is the case of the stuck-at 0 of the inputs / output of an OR gate, or of the stuck-at 1 of the I/O of an AND gate, or the stuck-at opposite values of the input and output of an INVERTER. These faults are *system equivalent*.

It is not sufficient to only consider the faults of an isolated component. Next, we must examine every single fault of the components of the circuit. A *complete diagnosis* (or *distinguishing*) *sequence* will have to classify all the

groups of faults that can be distinguished; each group is a set of *system equivalent faults*.

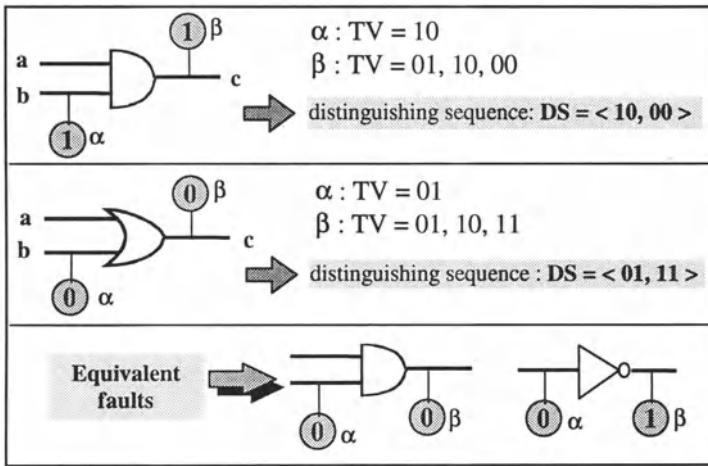


Figure 13.14. Diagnostic of gates

13.4.2 Study of a Small Circuit

Example 13.6

Let us consider once again the circuit of Example 13.1 (Figure 13.15) with a classic single stuck-at '0' or '1' fault model. We apply the following fixed diagnosis sequence: $DS = \langle 110, 010, 100, 011 \rangle$.

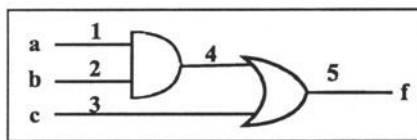


Figure 13.15. Diagnosis of a small circuit

The faults detected by these 4 vectors are indicated by Table 13.1: thus, the input vector $(abc) = (110)$ detects the stuck-at '0' of the lines 1, 2, 4 and 5. The symbol '-' indicates that the vector detects no failure of the line. This table was obtained by application of the *path sensitizing method* presented in section 13.2. The considered fault model comprises 10 faults (stuck-at '0' or '1' of each of the 5 lines). The circuit is either faultless (this is noted by the symbol ϕ), or affected by one of these 10 failures; this gives 11 situations. The application of the vector $V1$ separates these possibilities into two groups:

final classes; thus if we remove any vector, some of the distinction classes that are obtained are no longer equivalence classes.

13.5 INFLUENCE OF PASSIVE REDUNDANCY ON DETECTION AND DIAGNOSIS

Now we analyze the influence of passive structural redundancy (Chapter 8) on the test of logic circuits. By definition, passive redundancy implies *undetectable* faults. From the user's point of view, this situation can appear satisfactory: he/she does not have to preoccupy himself/herself with faults that never disrupt the good working order of the product. Such an argument is false, as we will show by the use of some simple examples. First of all, the manual or automatic search for test vectors detecting undetectable faults is a cause of loss of time and money. Secondly, the majority of protection methods are based upon simplified hypothesis such as the 'single fault' assumption. Passive redundancy can mask the existence of faults that can be accumulated over time. This is called *fault masking*. Passive faults can:

- mask the detection of 'activable' faults (a fault being able to bring about a failure is not detected by the tester), as illustrated by Example 13.7,
- distort the diagnosis of 'activable' faults (the tester is mistaken when identifying the presence of a particular fault when it is another fault), as illustrated by Example 13.8.

Example 13.7

Let us assume that a non-detectable stuck-at 1 fault occurs in the circuit of *Figure 13.17* (fault noted α in the figure). Since it is not detectable, it can occur and then remain in the circuit for a very long time without posing any problem nor being detected at the time of the maintenance test operations.

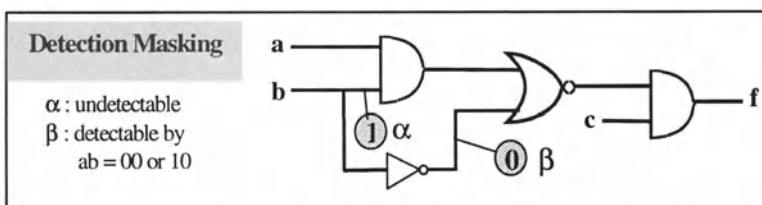


Figure 13.17. Detection masking

To understand the loss of detection, we consider a stuck-at 0 fault, noted β in the figure, which is detectable by two test vectors (00 and 10). We

suppose that we have chosen vector 10 for the maintenance test sequence. If this fault β occurs whilst α is already present, the result of the test is always ‘correct’ and therefore the fault β can no longer be detected by the test sequence. We say that *fault β is masked* by the passive fault α .

Example 13.8

In order to understand the loss of diagnosis, we propose the circuit in *Figure 13.18*. As for the previous example, we assume that a passive fault, noted α , is present in the circuit, perhaps for a very long time. Now let us consider two detectable stuck-at 0 faults, β and γ , which are distinguishable by the diagnosis sequence $\langle 11, 01 \rangle$. If the fault β occurs whilst α is already present, this sequence erroneously diagnoses the presence of γ instead of β !

Note. This situation corresponds to the presence of two single faults, which violates the classic single fault assumption. Actually, the first non-testable fault α appears at time t , and, from this moment, the probability of having a second failure increases with time. Such situation is quite possible.

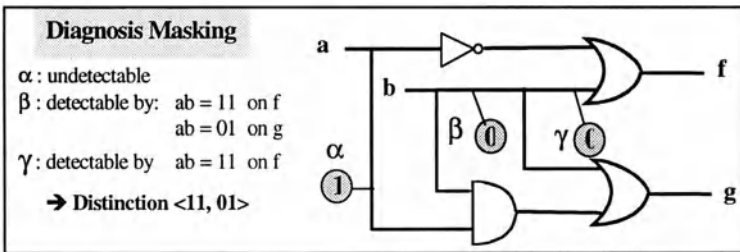


Figure 13.18. Diagnosis error

13.6 DETECTION TEST WITHOUT ERROR MODEL. APPLICATION TO SOFTWARE

13.6.1 The Problem of Structural Test without Error Model

In electronics, an error model defines error classes (for example that of the stuck-at) due to various causes: design errors of the components, hardware faults due to ageing, etc. Each class represents a perceptible physical reality that has been established by experience/experiment (notion of pertinence of the models). Furthermore, a model identifies generic errors appropriate to the technology and not to the product studied. For example, the stuck-at fault can occur in a gate constituting the product, whatever the functionality offered by this product. In the case of software technology,

such generic error classes are generally not considered, since they are not acknowledged as representatives of situations that are commonly encountered.

Nevertheless, the methods of structural testing for software are based upon types of elements that the test sequences must activate. For example, if we consider statements as the type of element, a structural test sequence will look to execute all the statements of a program. If this is the case, we will allude to a 100% coverage rate. As previously mentioned, the coverage rate of the sequence will be the number of elements (of the considered type) that are activated by the execution of the sequence, divided by the total number of elements of this type in the program.

Even if the terminology (*coverage rate*) is identical to that employed in electronics, the sense given in the software domain is different. In electronics, this level represents the percentage of the errors in a class (fault model) of which detection is guaranteed; in software, no precise link has been established with technological faults. We can say that the higher the value of this level, the more important the probability of detecting possible errors will be, since the sequence will have activated more elements. However, a level of 100% can be associated with a sequence that does not really detect 100% of the present faults. Even this sentence can be contested, since we do not know the set of possible faults, and thus the reference base that helps us to define the percentage of actual fault coverage.

Let us consider the following function:

```
function F(A, B: in float) return float is
begin
    return (A+B) / (A-B);
end F;
```

The execution of $F(5,3)$ provides a level of 100% when 'statements' are the type of elements considered, whereas this function contains an activable design fault raised by calling F with identical values ($A = B$) if this case has not been included in the specification.

Thus, the fact that all the statements have been executed increases, in an unjustified way, the trust that we could have. On the other hand, the fact that all statements have not been executed increases, in a justified way, the mistrust that we should have. Actually, the fact that it may not be possible to execute some of the statements in the program frequently uncovers a design fault. This situation however can be intentional, for example, as a result of library usage in which only some of the functionalities are voluntarily employed, or because of the presence of redundant mechanisms that were introduced for tolerance reasons, which are only activated when errors occur.

In order of implementation complexity, we consider as type of elements: statements, branches and paths, and conditions. We will indicate the increasing degree of the verification of these methods that are besides

generally demanded according to the increasing degree of dependability required. In the case of the requirements of the DO-178B standard applied to avionics systems, the following is appropriate. The software at level 1 (not critical) does not require the use of structural testing, those at level 2 require a *statement test*, for level 3 a *branch test* is necessary, and at level 4, which is the highest, a *conditional test* is required.

13.6.2 Statement Test

The *statement test* sequence must provoke the execution of all the statements of the program. Let us consider the following fragment of program:

```

if X <= 0                -- 1
  then  X := -X;
        Y := 2;         -- 2
  else  X := 1-X;
        Y := 1;         -- 3
end if;
if X = 0                 -- 4
  then  X := 1;         -- 5
  else  X := X+1;       -- 6
end if;

```

The figures following the '--' sign are comments which will be used in our analysis.

If we execute this program fragment with the value -3 and then +1 for X, we obtain coverage of 100%. Let us indicate that there are tools that are integrated into the execution environments of computer languages that can save and therefore highlight the statements executed.

Generally, we do not initially try to establish a specific activation sequence. We use the input sequence of the functional test sequence already defined for functional verification. The fact of not considering the output values shows that the software structural test is not used to detect failures of the product. If some of the statements of this program are not executed by this sequence, then that proves:

- either that the functional test sequence was incomplete and that certain behaviors were not evaluated,
- either that the design choices have lead to treat in a singular manner the behaviors that are considered as functionally equivalent,
- or that there is redundant code, for example to implement mechanisms for fault tolerance.

In order to illustrate the second case, let us consider that we have two functions `Small_F` and `Big_F` at our disposal. Both of these functions

calculate the same result F according to two different methods (or algorithms), respectively well adapted to the case where the value of the parameter X belong to the interval $[0.0, 1.0]$ or if it is greater than 1.0. Therefore, the designer will undoubtedly write:

```
if X <= 1.0   then F := Small_F(X);
              else F := Big_F(X);
end if ;
```

If the product (the complete program) behaves differently if $X \leq 7.5$ and $X > 7.5$, then the functional test sequence can execute the program with $X = 7.0$ then 8.0 and finally 7.5 for the 'limit tests'. In this case, the statement 'F:=Small_F(X);' will never be executed. Hence, the implementation of Small_F can contain a fault that we must try to detect. This example thus shows one of the interests in structural testing.

13.6.3 Branch & Path Test

We have just shown that the test of statements constituted a contribution to the evaluation of the functional test sequences. However, the supplied coverage level is of little significance. Let us consider the following extract:

```
if X>0.0 then
    ...
    ...   99 statements in sequence
    ...
    else  ... 1 statement
end if ;
```

If we execute it with $X = 1.0$, we provoke the execution of 99 of the 100 statements. The level of 99% seems very satisfying, whereas evidently, we have only tested half of the situations: ' $X > 0.0$ ' has been examined but not ' $X \leq 0.0$ '. In fact, the statement test makes the hypothesis of the independence of the statements, which is not the case in this program. In the previous example, the execution of the first statement automatically implies the other 98 statements that are in sequence.

The statements in sequence are now grouped together by branch and only the branches are examined.

Figure 13.19 shows a representation of the branches from the extract of the program that is provided in sub-section 13.6.2. The statement test explained in the previous sub-section leads to the state paths that follow:

- $X = -3$: 1, 2, 4, 6, 7,
- $X = +1$: 1, 3, 4, 5, 7.

Thus, all the branches are covered (2, 3, 5 and 6) by this *branch test*. This coverage level is more significant than the previous one, since it takes

the sequential dependencies between the statements into account. However, this test does not consider the dependency (or non-dependency) between the branches. For example, the state sequence 1, 2, 4, 5, 7 is not activated by the 2 attempts ($X = -3$ and $X = +1$) when it could correspond to a particular behavior that potentially leads to a failure.

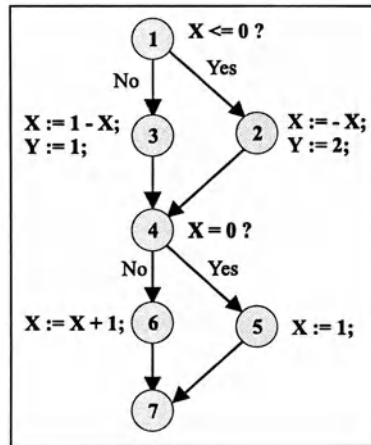


Figure 13.19. Branches of a program extract

The *path test* is devoted to the study of paths. We are therefore looking for a test sequence that provokes the execution of all the control graph paths of the program. For the previous example, the 4 possible paths are: 2 then 5, 2 then 6, 3 then 5 and 3 then 6. The sequence $X = -3$ then $X = +1$ therefore covers only 50% of the paths (2 out of 4). The *path test* thus seems preferable. However, it must be practiced with care.

First of all, all the *control paths* are not necessarily *execution paths*. In the previous program extract, if we replace the second test $x = 0$ by $x = -1$, then the path 1, 2, 4, 5, 7 will never be executed. This is because if $X \leq 0$ the execution of $x := -x$ gives X a positive value, which will therefore never be equal to -1 . In fact, the conditions for branching are not independent. If numerous relations exist between the conditions and the branches, then the actual number of execution paths is relatively low.

Secondly, the duration of such tests can be intractable, in particular if the program has loops and nested control structures. Actually, the number of paths increases in a combinatorial fashion. An article presented an example of a program that contains 2 'while' loops of which we assume that there is a maximum of 12 iterations, 2 'case' statements (equivalent to 'switch' in the C language) and one 'if' statement. For this program, we assume that the execution of a path requires 1 nanosecond. The study showed that an exhaustive path test would require 4000 years.

13.6.4 Condition & Decision Test

We use the term *condition* for a Boolean expression which does not contain any Boolean operator ('and', 'or' and 'not'). 'A>B' where A and B are float variables is such an example. We use the term *decision* for a combination of conditions (or of decisions) with the use of Boolean operators. 'A>B and B>C and not Cond' where Cond is a Boolean variable is an example of a decision.

The *Condition/Decision test* (C/DC for 'Condition/Decision Coverage') is close to the structural *toggle test* seen in Chapter 12 since it considers that:

- the decisions must take the value *True* and *False* at least once,
- the conditions must take the value *True* and *False* at least once,
- the input and output points of the components (subprograms, etc.) must be executed at least once.

This last condition aims at detecting components that are not used or in which the execution never terminates.

Let us consider the following program extract:

If A=B and C2 and D>3 then ...

We have 3 conditions (A=B, C2, and D>3) and one decision ('A=B and C2 and D>3'). Let us consider *Table 13.2* that contains the values of the 3 conditions (the first three columns). We deduce from these first 3 columns the values of the Decision. In this situation, we obtain a total coverage concerning the conditions and the decisions that take the values *True* and *False*. Thus, to acquire a test that gives a coverage of 100%, it is sufficient to take the values *a*, *b*, *c2* and *d* for variables A, B, C2 and D such that *a = b*, *c2 = True* and *d > 3*, then such that *a != b*, *c2 = False* and *d ≤ 3*.

Condition/Decision test does not try to obtain all the possible Boolean combinations but:

- to cover all the branches by assigning the values *True* and *False* to the decisions,
- to cover all the situations of the conditions.

A=B	C2	D>3	=> Decision
True	True	True	True
False	False	False	False

Table 13.2. Condition & decision test

The *C/DC* is the *Condition/Decision Coverage*. It measures the coverage, taking the three previous element types (conditions, decisions, I/O) into account. The *MC/DC test* for ‘*Modified Condition/Decision Coverage*’ adds a fourth requirement to the 3 previous ones:

each condition in a decision must be shown to independently affect the result of the decision.

In the previous case, the 3 conditions and the decision have taken the two possible values *True* (respectively *False*) but not independently: ‘ $A = B$ ’ and $C2$ and $D > 3$ were simultaneously *True* or *False*, which gave a *True* decision (respectively *False*). To achieve the previous requirement, we must modify a single condition at a time so as to show the impact on the decision. An example is developed in Exercise 13.14.

The achievement of the structural test sequences, in particular of the *C/DC* and *MC/DC* type, is difficult; since it must be able to control the modification of the values of the internal variables from the external inputs of the program. This is a controllability problem that has already been highlighted.

13.6.5 Finite State Machine Identification

Structural testing methods used to check programs are not based on explicit fault models. After the presentation of the statement, branch, path, decision and condition testing techniques, we must say that implicit fault models exist. Indeed, the *control flow* of a program can be expressed by a Finite State Machine (FSM) whose transitions are labeled by decisions and inputs. The introduced structural coverage rates in fact aim at measuring how a changing in the control flow FSM due to any fault can be detected.

Consequently, test sequence generation can be processed by finding inputs/outputs which identify the control flow FSM.

13.7 DIAGNOSIS WITHOUT FAULT MODELS

13.7.1 Principles

The diagnosis method presented in section 13.4 assumes the knowledge of a possible fault or error model. For each of the faults or errors of these models, the failures of the system are deduced in the same way as the trees that associate each failure with primary faults or errors. Thus, the ‘experimental approach’ also called ‘empirical associations’ assumes a priori the knowledge of the faults, failures and their relationships. Then, when a failure occurs, the potential faults are then examined.

Such techniques are not relevant when software technology is concerned, as the lists of faults or classes of faults that may actually exist cannot be exhaustively defined. To handle such a situation, the 'model-based diagnosis' technique does not assume a priori any knowledge on the software failures and faults. It takes a posteriori the occurred failure into account, analyzing two system models:

- what is expected, such as the system specification,
- what exists, such as the software structure (design models, program, etc.).

A failure violates an expectation whose causes must be found in the existing system structure.

A diagnosis method aims at guiding the engineers during the diagnosis process. Taking the previous pieces of information into account, this process helps to find the fault being at the origin of the failure.

The proposed diagnosis method is divided into four steps:

1. highlight the error,
2. elaborate hypotheses on the causes,
3. confirm these hypotheses,
4. verify these hypotheses.

In the following sub-sections, each step is defined and its practical use is studied. We will show that the main difficulties in handling these steps come from the lack of knowledge on the actual system functioning before the failure occurrence. To obtain such pieces of information, checks of the system functioning are processed at run-time and data are stored. These two activities are known as *system instrumentation*. The checks aim at confirming the correct operation of the components or to signal error raising at run-time. The data will be useful at diagnosis time to obtain the same conclusions, improving the knowledge on the actual functioning. The *instrumentation technique* will be introduced in Chapter 16. In this section, requirements on the instrumentation, which makes the handling of each step easier, are introduced.

Let us note that this diagnosis approach can be applied to any product (hardware and/or software) modeled at system level.

13.7.2 Highlight the Erroneous Situations

The first step consists in highlighting and studying the assumed failure from which we may conclude that a fault exists. This phase is important, as the engineer must be sure that a diagnosis is necessary. Two reasons stop the diagnosis process.

First reason. The analysis establishes that the detected erroneous situation comes from a defective use of the system. For instance, the user or the environment had previously lead the system in a state where occurring interactions cannot be handled as such a situation was not specified. This conclusion does not mean that no action has to be undertaken. Without doubt we will have to specify the contents of the user manuals so that the interactions offered by the system are more explicit. The specifications of the system can also be modified to improve the ergonomics so as to make the understanding of the state of this system and of the actions authorized clearer. Some mechanisms, which increase the robustness of the system to the undesired interactions, will have to be specified as well. However, these actions do not concern the search for a fault, since the system conforms to its specifications.

Second reason. The analysis highlights an incorrect instrumentation. We encountered systems whose functioning was correct but a failure was detected due to an erroneous instrumentation such as a too restrictive assertion on its expected functioning.

The study of the erroneous situation is very important, as it also specifies the problem to be solved. The conclusions of such a study will be the basic information handled during the following steps. In particular, this study must specify the circumstances of the failure by detailing the state or the sequences of states of the environment which led to the failure. This study must also provide an image of the internal state of the system, at that moment. This preliminary work is essential, since for many software systems the main difficulty is rediscovering the precise context of operation for the system. This context often corresponds to some very particular situations that were not imagined by the designers.

The operation configurations can be deduced from the observations that are made by the user. This information is often not precise, since the user does not pay special attention to it (he/she does not expect a priori any failure when it occurs). For these reasons, we recommend the planning of a data saving mechanism that characterizes the state of the operation requests, or the history of these requests in the case of a sequential system.

The context of the software system also includes the internal state of the system. If the identity of the current statement is useful, other pieces of information are useful as well. It concerns the state of the components of the system (objects, variables, etc.). This information can be quantitative, that is the current values of the variables. This is obtained from saved data or deduced from them. This information can also be qualitative, that is the state of the correct functioning of the components. This correct functioning is confirmed by the implementation of detection mechanism for erroneous states, for example by means of assertions. These two types of information

are often correlated. For example, if we save the value of the variable that designates the index of the top of a stack, this quantitative information also provides qualitative information about the state of the stack: empty, full or between the two, Pushed when it was full, etc.

13.7.3 Elaborate the Hypotheses

From the erroneous situation, that is the failure and the occurrence context, hypotheses on the causes are examined. Taking the program structure into account, the engineer searches for the states previously processed by the program to detect an internal erroneous one.

The main difficulties are quantitative as well as qualitative.

Quantitative difficulties. The program contains numerous statements whose backward recovery induces numerous possibilities. The engineers must be able to select a few numbers of them to continue their analysis. For instance, to handle backward simulation from the last statement of the following program extract, the engineer needs to know if the ‘then’ or ‘else’ part was processed.

```
if A>B then . . . ;
        else . . . ;
end if;
A:=A+1;
```

Moreover, only in few cases the execution of a statement possesses a single antecedent. For example, if we know the value of A after the execution of $A:=A+1$; , we can deduce the value of A before this execution (A has the value $A-1$). Generally, classes of possible values must be manipulated. For example, if the branch `then` was definitely executed, the variables A and B can previously contain all the values such that $A>B$.

Qualitative difficulties. Whereas the program provides structural information (statements are assembled by control structures), the elaborated hypotheses must concern more abstract notions such as “this subprogram provided a wrong result”. A link between these two types of information has to be maintained throughout the analysis. For example, the knowledge of the values returned by the parameters allows us to make a conclusion about the subprogram erroneous behavior.

At the end of this process, the provided hypotheses are program states whereas the fault that causes their occurrences at run-time is a structural notion. For instance, a constant value is incorrect or the Boolean condition is $A>=B$ instead of $A>B$, etc. are structural notions. As no realistic fault models exist for the software technology, the deduction of a fault from a given erroneous state cannot be done automatically. However, to make this last step easier, the elaborated hypotheses must be close to the actual fault.

13.7.4 Confirm the Hypotheses

The previous backward simulation leads to multiple hypotheses. Most of them are only potential. The confirmation stage of the hypotheses endeavors to show that an error hypothesis effectively leads to (or does not lead to) the erroneous situation. A direct simulation performs this research. We execute the system from the hypothesis.

The first difficulty comes from the fact that the information that acts as the starting point of the simulation is often expressed in constraint form: f does not return a correct result or the value of X is less than 1.0. Thus we have to perform a simulation that uses qualitative data (the first example) or uncertain data (the second example).

Here again, the program complexity makes high-level simulation necessary. For example, we will not execute a subprogram but assume that its execution provides a correct result. This assumption must be argued about. The instrumentation is therefore useful once more. For example, thanks to the checks that were carried out on-line, or from saved information, we deduce a certainty on the correction of the subprogram.

Whatever the simulation level we consider, the structure of the program describes multiple processing possibilities. For example, an *if then else* offers two choices. Similarly, it can be necessary to know the effective number of iterations that are executed by a loop. Pieces of information must be available to reduce the choices.

If the simulation handling does not lead to the assumed erroneous situation, the studied hypothesis is wrong. To reduce the time spent reaching this conclusion, the incoherent situations that come from the hypothesis should be detected as soon as possible. In particular, the simulation is stopped if it leads to states that could not occur. The instrumentation data are one again useful to detect such a situation. For example, if the hypothesis leads to show that the program requested the opening of a valve or affected a variable to some value, while these facts did not occur, this contradiction shows that the hypothesis is false. This conclusion is obtained without processing the simulation from the fault up to the failure.

13.7.5 Verify the Hypotheses

The previous analysis concerns one erroneous situation and the recent past of the system life: from the hypothesis to the error. However, the useful life of the system may be longer. So engineers use knowledge on this previous activity to confirm or not the assumed hypotheses. For instance, let us consider the following hypothesis: the subprogram P does not correctly calculate a result in certain circumstances. If P was previously called on

numerous occasions in the same circumstances without any problems, then this hypothesis is probably false.

When industrial applications are concerned, such a conclusion is unfortunately not always true. Even if the program structure cannot be changed at run-time, a lot of transient phenomena exist. For instance, they may due to hardware faults or software temporal problems. For example, an interrupt occurs whose handling consumed CPU time, increasing the execution duration of a given preempted task. The fact that the interrupt was raised during the task execution may be masked, thus making the understanding difficult of the task execution duration delay. Other hardware-software interactions are at the origin of such phenomena. For example, the presence of cache memory changes the execution duration of the executable code instructions. It has an effect on the application behaviors that may be perceived as hazardous. Here again, this cause is not understandable when the hardware platform characteristics are unknown.

13.8 MUTATION TEST METHODS

13.8.1 Principles and Pertinence of Mutation Methods

A *program mutation* consists in modifying its statements so as to obtain a new program called *mutant*.

The modifications used in practice are first-rate; they affect a single element. Various types of modifications have been proposed. They constitute fault models and concern different features.

- Constant values are changed by taking other values. For instance, in the case of a real constant, we shift the placement of the decimal point forward. For example, 3.14 is replaced by 31.4.
- Identifiers are replaced by others of the same type if the language is strongly typed (Ada, Pascal). In the case of languages that allow implicit conversions, such as the C language, an identifier can be substituted by another one, even if their types are different. For example, in the statement “X=Y;”, Y of type float is replaced by I of type integer.
- Operations are replaced by other operations that have operands and results of the same type, if the language demands the respect of types. This applies to arithmetic operators (+, -, *, etc.), but also to logic operators (or, and).
- The called functions are replaced by others that have the same number and the same types of parameters.

- The flow of control is modified. For instance, by adding a negation operator in front of the conditions that are associated with the branch tests (if) or loop tests (exit conditions). For example, “if ((A+B < C) and (A < D)) then . . . else . . .” is replaced by “if not((A+B < C) and (A < D)) then . . . else . . .”.
- Statements are removed.

The question of the representativeness of such fault models immediately comes to mind. These models generally correspond to no realistic faults. The promoters of this technique confirm this. It is actually unlikely that a designer obtains a correct program by means of a single mutation to the original program. However, this technique presents a number of interesting features. Even if the faults that are used by mutations are not realistic, the errors that these faults induce are generally significant. The mutation technique therefore allows us:

- to validate the error detection mechanisms by instrumentation of the code (see Chapter 16),
- to validate the test sequences.

This last aspect is developed in the next sub-section.

13.8.2 Mutation Testing Technique

The mutation technique is used to evaluate the test sequences. For each mutant, we check that the test sequence is capable of activating the fault introduced and propagating the induced error up to the outputs. If the mutation cannot be detected by the sequence, this sequence must be improved. However, this improvement is not always possible. First of all, the modification of the program by certain mutations does not lead to a fault. For example, let us consider the extract of the following program:

```
Number_of_Embedded_Passengers =
    Number_of_Registred_Passengers;
for (N=1; N < Number_of_Embedded_Passengers; N++)
    { . . . }
```

If a mutation transforms in the for loop, `Number_of_Embedded_Passengers` by `Number_of_Registred_Passengers`, then there is no fault detectable.

In other situations, the mutation effectively creates a fault, but the redundancies do not allow to activate an error or to detect it at the outputs. We will say in both cases that the *mutated program is equivalent* to the initial one, which means that its behavior is not influenced by the mutation.

Under these circumstances, we do not have to try to modify the test sequence since no improvement will be obtained. The performance of a test sequence is therefore measured by the number of mutations that are detected divided by the total number of applicable mutations, in excluding the mutations leading to equivalent programs.

Evidently, this technique is only tractable if we dispose of a tool that automatically injects the faults one by one, re-executing the test sequence and noting the detected faults. In addition, the types of mutations that are considered must be few in numbers. For example, a program containing 500 uses of the '+' operator will create 1500 mutants if we modify each use by '-', '*' or '/'. If a program contains 15 integer variables, each used 10 times; the total number of mutations is $15 \cdot 10 \cdot 14 = 2100$.

Finally, certain mutations give properties to the program that make the test difficult. Let us consider for example, the following extract:

```
for (I=1; I<=N; I++) { . . . }
```

with the mutation which replaces the identifier N by the identifier I , thus the execution of the loop can become infinite. This can cause an omission of a result, which cannot therefore be compared to the expected one.

In spite of these difficulties, this evaluation method of test sequences is quite appropriate, not as a result of the representativeness of the faults that are considered by the mutations, but as a result of the representativeness of the errors that these faults induce. Two types of experiments showed this.

- First of all, a test sequence was established, having a complete coverage of the set of mutations, for a preliminary version of the program. Then, this test sequence was applied to a second version that was developed from the same specifications, but by another person. Most of the faults that were present in this second version were detected.
- Secondly, the mutations can be applied two by two instead of one by one. We therefore show that the initial sequence also detects almost all of these situations. This experiment would aim to conclude that the sequence is capable of detecting the errors that come from complex faults, if those are considered as the combination of simple ones.

A similarity exists between the evaluation of the mutation software test methods and the evaluation of electronic test methods that have to detect the stuck-at 0 / 1 errors. We saw in the previous chapter that this evaluation was carried out by methods that are based on the simulation of faults, that is, by injection of faults. These errors have also been shown as significant. They are due to physical faults that are not taken into consideration since they are too diverse and complex for an observation of the system at logic level.

However, we have to repeat that the faults considered are not representative. Therefore the sequences obtained by the mutation testing

technique are useful in the detection of the presence of faults thanks to errors that they provoke, but not to the diagnosis of these faults. For example, a piece of a sequence could have been introduced to reveal the transformation of a '+' operator into '-'. If the sequence is applied to the initial program, no error should be detected. The detection of an error, by means of this sequence slice, in another version of the program does not allow us to deduce the presence of a fault which changes a '+' operator into a '-' operator in such a place in the program. In fact, the structure, i.e. the statements of the tested program, is undoubtedly very different to that of the original program.

The evaluation of a test sequence by the mutation technique requires the sequence to raise an error (controllability problem), but also to propagate it towards the outputs (observability). The *weak mutation testing* only possesses the first requirement. Indeed, it assumes the presence of internal detection mechanisms to signal the error detection, such as those of instrumentation that will be developed in Chapter 16. Only the controllability capability of the sequence is therefore evaluated. This technique is also useful to measure the performance of the detection mechanisms implemented in the application, for instance to recover errors by a fault tolerance mechanism.

13.9 EXERCISES

Exercise 13.1. Test of a small circuit

Consider the small circuit in *Figure 13.2*.

1. Apply the method seen in section 13.2 to find all the test vectors for each of the stuck-at '0' or '1' faults of this circuit. Draw the resulting *fault table* and comment the coverage efficiency of the different input vectors.
2. From this *fault table* deduce an optimal test sequence, i.e. a test sequence detecting all faults and having a minimal number of vectors.

Exercise 13.2. Test vectors detecting a fault

Consider again the circuit studied in Example 13.4. Apply the structural test method studied in section 13.2 to find all vectors testing the stuck-at **1** fault of line 11. The procedure here is more complex, since in stage 2 of the backward propagation several input values can satisfy the constraint. These possibilities must then be considered successively. Hence, a loop must be added to the basic procedure.

Exercise 13.3. Analysis of test procedures

Let us consider the circuit of *Figure 13.20*. We want to find, by a path sensitizing approach, one test vector which detect the stuck-at 0 fault of the output of gate *D*.

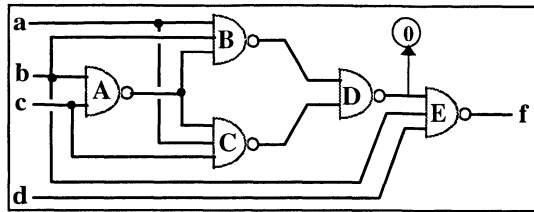


Figure 13.20. NAND-gate circuit

1. The following procedure is proposed:

- All lines are set to 'x' (unknown) state.
- Input *a* is set to '0', and a propagation (simulation) is performed towards the fault location: $B = 1$, $C = 1$, $D = 0$, so the fault cannot be activated; hence we make a backtracking in the input assignment.
- Input *a* is switched to '1', and a propagation is performed that brings nothing, as *B* and *C* stay unknown.
- Input *b* is set to '0', and a propagation is performed: $B = 1$.
- Input *c* is set to '0', and a propagation is performed: $C = 1$, so the fault cannot be activated; hence we make a backtracking in the input assignment.
- Input *c* is switched to '1', and a propagation is performed: $C = 0$, $D = 0$, so the fault is activated as an error *e*.

Analyze this procedure and complete it in order to find one test vector.

2. The following procedure is proposed:

- All lines are set to 'x' (unknown) state.
- To activate the fault, *D* is set to 1, and we backtrace this information towards a primary input: one input of gate *D* must be set to '0'
- We choose to set *B* to '0', and we backtrace this information: all inputs of gate *B* must be set to '1'.
- We choose first the hardest case, i.e. to assign *A* to '1', and we backtrace this information.
- We chose to set *b* to '0', and we perform a forward propagation towards the fault of this information: $B = 0$, so the assignment is inconsistent; we abandon this path and try another one with the assigned values of the primary inputs.

- Gate C is set to '0', this information is backtraced: all inputs of gate C must be set to '1'.
- Input a is set to '1'.
- Input c is set to '1'; now the fault is activated as an error.
- We want to propagate the error through gate E : there is inconsistency.
- We make a backtracking in the primary input assignment.
- Input c is switched to '0', and we perform a propagation: $C = 1$, so there is inconsistency, and we continue the backtracking on input b , c being now set to 'x' again.

Complete this procedure in order to find one test vector. Can this procedure be improved?

Exercise 13.4. Fault coverage of a test vector

1. We apply to the circuit of Exercise 13.2 the input vector $(a\ b\ c\ d) = (1\ 0\ 0\ 1)$. Find by structural analysis all the faults covered by this vector.
2. Is the preceding vector a good test vector? Analyze the structure of the circuit in order to find test vectors having the highest test coverage (best number of detected faults).

Exercise 13.5. Diagnosis of a circuit

Let us consider again the circuit of the previous exercise.

1. Look for a simple test sequence that at best distinguishes the three faults in the group $\{2^0, 5^0, 11^1\}$.
2. Draw the diagnosis tree of the following test sequence;
 $TS = \langle 1000, 1001, 0110 \rangle$.
3. Is this sequence significant?

Exercise 13.6. Complete diagnosis of a small circuit.

From the fault table obtained in Exercise 13.1, deduce a minimal sequence which performs the best diagnosis testing.

Exercise 13.7. Logical test of some faults of a full-adder

This exercise invites you to test the full-adder of which the logic diagram is given again in *Figure 13.21*.

1. Find all the input vectors which test the stuck-at '0' fault α in the figure.
2. This fault has already been studied in Exercise 5.2 of Chapter 5. In what way is this functional approach different to the present structural *path sensitizing* approach?

3. A functional fault transformed the two EXCLUSIVE OR gates into IDENTITY gates. Is it possible that a test vector that was found in question 1) can test this fault, which is not a stuck-at fault? Why?

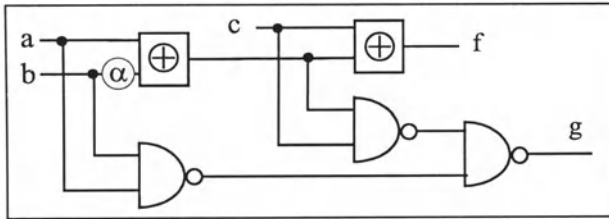


Figure 13.21. Full Adder

Exercise 13.8. Functional test and toggle test of a full-adder

1. Find a very simple functional sequence of two vectors to test the full-adder of the previous exercise. Find the stuck-at faults detected by this sequence.
2. Complete the previous sequence in order to obtain a *toggle* type test sequence (such a sequence was found in Exercise 12.2 of Chapter 12).
3. Complete the previous sequence to test all the stuck-at faults of the circuit.

Exercise 13.9. Test of a structured circuit

Two full-adders (FA1 and FA2) are connected in series as shown in Figure 13.22. This structured circuit adds two 2-bit numbers, $(a_1, a_2) + (b_1, b_2)$, and produces a 3-bit result, (S_1, S_2, S_3) . An input carry c_0 (set to 0 in our case) can be used to put in cascade several circuits, in order to constitute an n -bit adder.

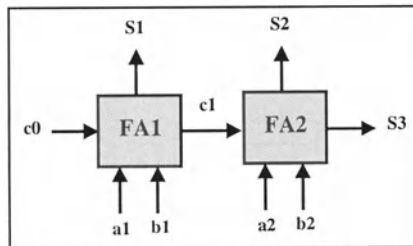


Figure 13.22. Structured adder

We suppose that each full-adder is completely tested by the following test sequence made of 5 input vectors (a, b, c) : $\langle 001, 010, 100, 110, 011 \rangle$.

1. Is it possible to apply such a sequence to each module from the primary inputs and outputs?
2. From this study, deduce a complete test sequence that ensures the detection of all faults in this circuit.

Exercise 13.10. Diagnosis study of the full-adder

Draw the diagnosis tree of the input sequence $\langle 000, 010, 111 \rangle$ applied to the full-adder of Exercise 13.7. Is this sequence a good diagnosis sequence?

Exercise 13.11. Complete test sequence of a circuit

A very simple logic circuit of 3 inputs (a, b, c) and 1 output (f) is depicted in the following figure.

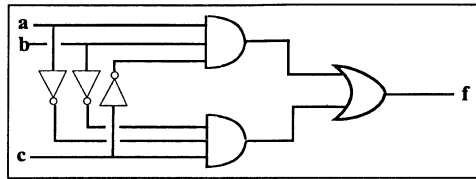


Figure 13.23. Complete test of a circuit

Find a test sequence that detects all the single stuck-at 0/1 failures of the inputs / outputs of the gates in this circuit.

Exercise 13.12. Redundancy analysis

We consider the circuit described by Figure 13.24.

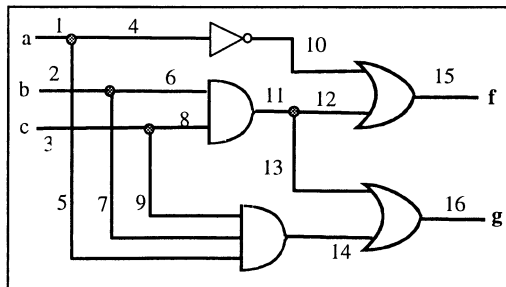


Figure 13.24. Redundancy analysis

1. We want to determine the passive structural redundancies of this circuit. You will address this question by two different approaches:
 - by an algebraic Input-Output study,

- by drawing the fault table of this circuit.
2. Find the conditions leading to the masking of fault detection, and the conditions leading to the masking of diagnosis.

Exercise 13.13. Structural testing of a program

A program, written in C, is aimed at regulating the temperature of the engine of a car in the range [0°C, 90°C] (function named 'regulator').

1. Analyze the functioning of this program.
2. Define some functional test vectors.
3. Analyze the coverage of these test sequences.

Program:

```
int modify_temperature(int temperature, int action,
                      int duration)
/* int temperature, action, duration; */
{
int final_temperature, t;
switch (action)
{
case 1: /* Heating */
    final_temperature = temperature;
    for (t=0; t<duration; t++) final_temperature++;
    return(final_temperature);
case 2: /* Cooling */
    final_temperature = temperature;
    for (t=0; t<duration; t++) final_temperature--;
    return(final_temperature);
default: final_temperature = temperature;
        return(final_temperature);
}
}
#define heat_a_little 10
#define heat_much 20
#define cool_a_little 10
#define cool_much 20
/* ***** */
int regulator(int initial_temperature, int
heating_state, int fan_state, int variation)
/*
    XX_state    = 0 faulty
                = 1 OK
    variation    = 0 a little bit
                = 1 much
    return      = the final temperature if the
                regulation is possible
                = -3000 if the regulation is impossible
*/
```

```

*/
{
int temperature, final_temperature;
if ((initial_temperature > 0) &&
    (initial_temperature <90))
    return(initial_temperature);
else {
    if ((initial_temperature <0) &&
        (heating_state==1))
        {
        temperature = initial_temperature;
        while (temperature <0)
            {
            if (variation==0)
                temperature =
modify_temperature(temperature, 1, heat_a_little);
            else temperature =
modify_temperature(temperature, 1, heat_much);
            }
        final_temperature = temperature;
        return(final_temperature);
        }
    if ((initial_temperature > 90) && (fan_state==1))
        {
        temperature = initial_temperature;
        while (temperature > 90)
            {
            if (variation==0)
                temperature =
modify_temperature(temperature, 2, cool_a_little);
            else temperature =
modify_temperature(temperature, 2, cool_much);
            }
        final_temperature = temperature;
        return(final_temperature);
        }
    if ((heating_state == 0) || (fan_state == 0))
return(-3000);
    }
}

```

Exercise 13.14. MC/DC testing of a program

We consider the following fragment of a program:

```
    If (A=B and C2 and D>3) then Action; end if;
```

Define the sequence of values that must be taken by the conditions and the decision so as to realize the *MC/DC* test.

FOURTH PART

FAULT TOLERANCE MEANS

After having considered the fault avoidance techniques in the third part of this book, we now tackle the dependability means that are relative to fault tolerance.

First of all, we dedicate Chapter 15 to *Error Detecting and Correcting Codes*, which constitute a fundamental base for most techniques used to design fault-tolerant products. Thus, this chapter aims at introducing the notions developed after.

Then, Chapters 16, 17 and 18 present in a progressive way the principles, interests, and issues of fault tolerance: on-line testing systems, fail-safe systems and fault-tolerant systems.

Finally, in Chapter 19 we compare the various approaches that have been studied throughout the book to synthesize their contribution to dependability improvement.

Chapter 14

Design For Testability

Faced with the ever-increasing complexity of testing, scientists have developed several methods and techniques to make test pattern generation easier, as well as to decrease the length of these sequences in order to reduce the testing duration, by acting on the product design.

14.1 INTRODUCTION

14.1.1 Test Complexity

The prodigious historical evolution of computer technology, concerning hardware as well as software, has led to products of increasing complexity. According to the Moore's empirical law, this complexity has been multiplied by 2 every 1.5 years! Current integrated circuits have millions of transistors, hundreds of I/O pins, and clock frequencies greater than one gigahertz, with feature sizes around 100nm. The cost of today automatic test equipment is several million dollars, and it might be growing exponentially to cope with the next generation of ICs. According to an SIA (Semiconductor Industry Association) report, the cost of testing future chips using conventional methods should rapidly be superior to the cost of manufacturing these chips. The same dilemma is developing with software applications, where the complexity is also increasing considerably in embedded applications (millions of statements). For this technology, the cost of testing increases exponentially; it is now frequently higher than the design cost.

It follows that the testing of products integrating both electronic components and programs has become rapidly prohibitive, even impossible, as illustrated by *Figure 14.1*.

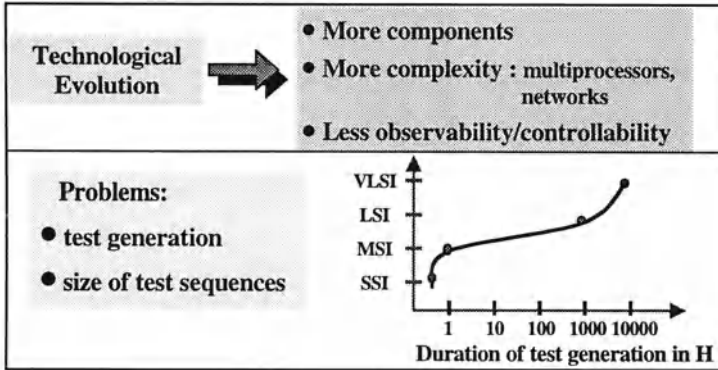


Figure 14.1. Test complexity

The *off-line test*, which was analyzed in the previous chapter, poses two kinds of problems:

- At the *test pattern generation level*, that is for the search of a test sequence:
 - Which method must be used so as to obtain a test sequence?
 - What is the cost of this search, in terms of study duration and of invested finances?
- At the *test application level*:
 - What is the cost of the testing equipment?
 - What is the length of the sequence (point which is connected to the duration of the test)?

Systems that are 'easily testable' bring fundamental contributions to answer these questions. After explaining the principles of such systems (subsection 14.1.2), we will specify the means that allow the modification of the designed product (section 14.2), then that allow the design of easily testable products (sections 14.3, 14.4 and 14.5). To finish this chapter, we will show the evolution from *off-line* to *on-line testing* (section 14.6).

14.1.2 General Principles of Design For Testability

14.1.2.1 Testability

On the basis that testing is a difficult challenge, we pose the question:

Is it possible to design and manufacture a product that is easy to test?

First of all, let us specify what we mean by 'easy to test'. This concept, involves *testability* criteria and underlies two distinct yet linked properties:

- the easiness with which we can **generate** a *test sequence* satisfying some quality *properties*,
- the easiness with which we can **apply** this test sequence to the product.

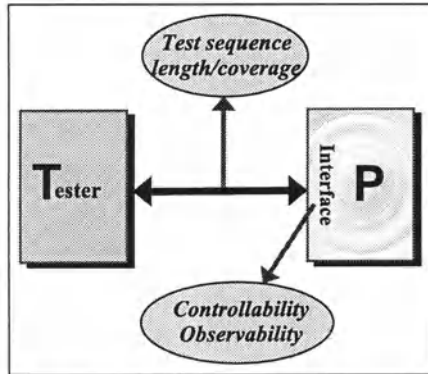


Figure 14.2. Easily testable systems

Test quality

The general *test* context comprises the *product* expressed by a *model*, a *fault model*, and a *test sequence* applied to this product by a *tester* (Figure 14.2). The *quality of the test sequence* is quantified by two attributes already presented:

- the *length of the sequence* which influences the application easiness (duration of the test experiment); this attribute often assesses the test generation easiness;
- the *coverage* of the sequence which measures the test efficiency to detect faults of the considered fault model.

Test generation aims at defining a test sequence satisfying the test quality requirements expressed before. These attributes are useful to choose a suitable test method. For instance, a random test sequence is easy to obtain, but its length is generally prohibitive to reach a high coverage rate.

Testability measurement

Now, let us consider the testability by examining the product model. The easiness with which we generate the test sequence depends on the complexity of the product (number of elementary components, number of internal states of sequential systems, etc.), and on the way it has been structured into interconnected components.

The most important attributes that can be used to measure the testability of a product are the *controllability* and the *observability*.

- **Controllability** is the easiness with which we can influence the product functioning so as to apply a given stimulus on an internal component from the primary inputs (ease of producing an arbitrary signal at the input of a component by exercising the primary inputs of the product).
- **Observability** is the easiness with which we can access (to get a measure) the internal variables and states from the exterior by the application of a stimulus (ease of determining at the primary outputs of the product what happened at the output of a component).

These characteristics are linked to the structure of the product, that is to say, to the way in which it is organized as a system, into interconnected modules, possibly in accordance with a hierarchy.

Every increase in controllability and observability facilitates the test: this is the basis of all the methods that are presented in this chapter. Indeed, this increase allows us to change a fault into an error (*controllability*), then this error into a failure of the product (*observability*). *Figure 14.3* illustrates the lack of controllability and observability of two poorly testable circuits.

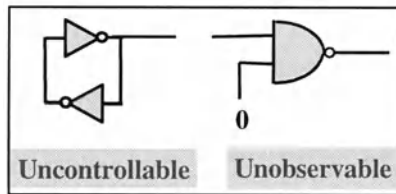


Figure 14.3. Lack of controllability and observability

Several different groups of reasons can explain why a given product has a low controllability or observability. First, the design of the product has not produced an optimal structure (bad method or bad use of the method). We have already seen that some redundant components cannot be tested (untestable parts). Unfortunately, these redundancies are extremely frequent (in the form of reconvergent fanout structures for example) and difficult to eliminate during the design process. However, some redundancies can have voluntarily been introduced during the design for good reasons! For example, gates are frequently added to combinational structures in order to eliminate the occurrence of glitches; the anti-glitch circuit is then redundant, and the added gate cannot be tested! We will study some examples of gate circuits as exercises at the end of this chapter.

Even if the design is optimal according to redundancy (no redundant components), the resulting structure can be difficult to test, as controllability and observability properties are not taken into account by most design methods.

It is important to note that the testability of individual components or parts of a system do not guarantee that the entire system is testable when these components are working all together. Hence, as we already noticed, the ‘unit testing’ aimed at testing individual components is a necessary step in the process of testing a complex system; however, are also required ‘integration testing’ procedures, which are much more difficult to implement and manage.

Several methods, not detailed in this book, have been proposed to evaluate the testability of a given hardware or software system. They are essentially based on structural analysis of the system, in order to deduce *qualitative estimators*. At gate level, the number of gates (NG) is a rough approximation of testability measurement of a circuit. Adding the number of inputs and outputs (NIO) leads to a better measurement ($T = NG / NIO$). Much more accurate methods to evaluate the testability of a gate circuit have been proposed, and some of them have led to evaluation tools. All of them try to estimate the controllability and the observability of the circuit. These methods have led to industrial analysis tools, such as SCOAP.

In software, a popular example is related to *software quality metrics*. The quality is measured according to several parameters; for example, the Halstead software metrics is based on 7 parameters: vocabulary, size, volume, difficulty, effort, errors estimated, and testing time.

Test application

We must note that the easiness of the *test application* can involve very technical aspects like the interconnection between the tester and the product under test: external physical connections (connectors, pads, etc.), probes or other sensors used between the tester and the tested product. This aspect will not be developed in this book although it can pose major technical problems. For instance, the occurrence of an output considered as erroneous may not be due to a failure of the product under test, but due to a parasite induced by the test environment as well.

14.1.2.2 Design for Testability

The *Design For Testability (DFT)* strategy (sometimes called *Design For Test*) provides solutions which make the product testing easier. Some of them act on the structure of the product already designed so as to increase the controllability and observability, and hence facilitate test generation. Other ones act on the design of the product by integrating specific mechanisms which facilitate test application.

We identify four groups of techniques, as illustrated in *Figure 14.4*:

- *ad hoc approach*,

- *specific design for testability*,
- *Built-In Test (BIT)*,
- *Built-In Self-Test (BIST)*.

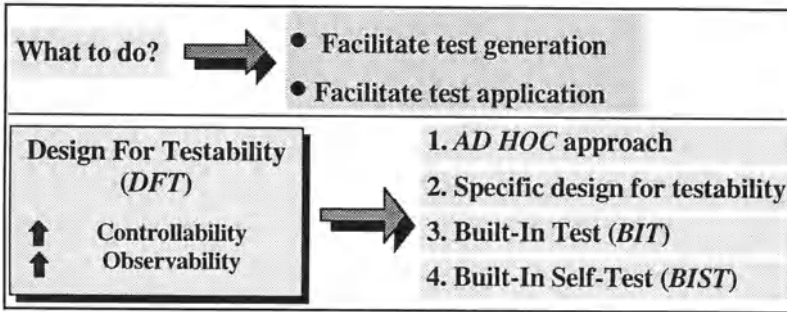


Figure 14.4. What can be done?

The first approach is very pragmatic: it uses guidelines and techniques that are applied either during or after the design stage. However, these rules do not have a profound impact on the structure of the designed system. Nevertheless, they bring a significant improvement to testability. On the contrary, the second approach requires a huge effort for specific design. The mechanisms implemented are therefore specific to each system class, which makes a unified presentation difficult. We will illustrate their principles with the help of two examples. Concerning the third and the fourth family of techniques, we do not question fundamentally the design, but we act mainly on the external interface with the tester. These techniques aim at improving the cost of the *off-line test* by integrating the functions of the tester into the product. The English term *self* in the last group of techniques can lead to confusion with the on-line *self-test* techniques which are presented in Chapter 16. Actually, all techniques presented here require the product to be halted during the test, whereas those presented in Chapter 16 operate in parallel with the normal function of the system. Nevertheless, we will encounter on-line implementations of the BIST techniques in high dependability systems.

These four families are analyzed in the following paragraphs. The ad hoc techniques are presented in section 14.2. Section 14.3 deals with specific design for testability methods. The *Built-In Test* approach is presented in section 14.4, and the *Built-In Self-Test* is considered in section 14.5.

14.2 AD HOC APPROACH TO DFT

The principle of the *Ad Hoc approach* is to modify the structure of the partially or totally designed system in order to facilitate access to it from the outside. Fundamentally, it is a pragmatic approach to control and to observe certain variables from the outside. It has frequently been referred to as ‘afterthought testability’. Naturally, these basic considerations should lead to more formalized and efficient methods that cope with testability at the early stages of the design.

In this section we will introduce the general design guidelines that bring new characteristics to the system, in order to make its test easier. To illustrate the application of these guidelines, we develop in sub-sections 14.2.2 and 14.2.3 two software mechanisms that facilitate the observability and the controllability: the *instrumentation* and the *exception mechanism*.

14.2.1 Guidelines

In this section we are going to briefly present seven groups of *guidelines* concerning the best practices for improving testability.

Initialization: *Add if necessary initialization devices*

These are reset signals for electronic components, or initialization procedures for software applications. These devices allow us to place the system into a known initial state, thus facilitating the test operation. This initialization is generally necessary to start the test, for instance, to test a sequential system. It can also be useful to bring back the system into a predefined state, after the application of a part of the test sequence. When such an initialization mechanism is not available, a long input sequence must often be included in the test sequence to reach a desired known state.

Modularization: *Partition the system into small modules loosely and explicitly coupled (or ‘divide and conquer’)*

A good guideline for system design advises the use of solutions for which the components are loosely and explicitly coupled. This expresses the fact that there are only a few dependencies between the components and that they are clearly visible. For example, the subprogram body has to call a small number of other subprograms (loose coupling). Furthermore, the values necessary for the execution of the subprogram or returned by its execution must be passed by parameters and not by global variables (explicit coupling). Such a practice facilitates the test, since we know precisely the small number of causes that influence the behavior of a component, (controllability) and the effects produced by this behavior (observability).

Example 14.1. Modular realization of a counter

Consider a 16-bit counter which has one input receiving asynchronous events (I), and 16 outputs (Q_i) expressing the number of input events; the circuit reacts to the negative edge of I . The test of this counter requires $2^{16} = 65536$ input vectors: obviously, this circuit is not easily testable. Now, let us partition this system into two cascaded 8-bit counters, and add a multiplexer between them, as shown in Example 14.1: when $Test = 0$, the circuit behaves as a 16-bit counter, and when $Test = 1$, we obtain two independent 8-bit counters. Hence, the test complexity is drastically reduced. Indeed, the two 8-bit counters can be tested in parallel ($I = IT$) with 128 vectors (pulses on I) with $Test = 1$; then, a last pulse on I with $Test = 0$ checks the link between the two sub-circuits.

Let us note that this solution has not increased so much the circuit complexity in terms of number of components.

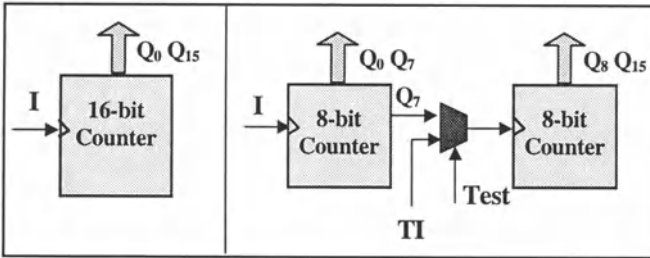


Figure 14.5. Modular realization

Hierarchy: Take into account the hierarchy of the design system by carrying out a bottom-up test process

A good practice for design consists in organizing the constituents into a hierarchy (electronic components, subprograms, data structures, etc.), so as to find in the designed system the different levels of abstraction introduced at the time of design. However, this approach, which can only be encouraged, diminishes the testability of the system. In effect, it is more difficult to modify the value of a variable of a component which is located deeply in a component tree structure, or to activate one of its features, than if all the components were at the same level. This situation leads to organize the test process into several steps:

- the *unit test* checks individually the low-level components,
- the *integration test* checks the components of a higher level in the tree,
- the *final test*, or *acceptance test*, checks the complete product.

Each verification level implements test techniques which assume that the lower level constituents are correct. Thus, if a subprogram uses 'sub-subprograms', we will try to cover all the paths in the subprogram body by considering the 'sub-subprograms' as *black boxes*.

The bottom-up test brings solutions to the controllability of hierarchical systems. Actually, the tests applied at a given level must activate the modules that make up the system at this level, without trying to exercise the internal structure of these components. These components are therefore considered as atoms.

Concerning the observability, mechanisms must allow us to automatically propagate the detection of an error across the design hierarchy. For software applications, we present in section 14.2.3 the propagation technique offered by exception mechanisms, which responds to this need.

Mastering of the sequential parts

1. Control the unknown states

Most designed systems do not use all the internal states which are, for example, defined by flip-flops in electronic design. If, for any reason, the final product enters one of these unknown states, it is important to know how it will react. If the behavior is 'trapped' into an uncontrollable sub-set of states, testing will become impossible. This situation corresponds to functional redundancy implied by the state coding.

Such rules are given by the companies designing integrated circuits (such as Programmable Logic Devices) or proposing tools to design them. In software technology, we encounter the same worry to master unknown situations. For example, when using a `case` statement, it is recommended to always insert the `default` condition to bring the system in a known state when an incorrect situation occurs. Let us note that in nowadays hardware developments, the use of VHDL or Verilog languages mixes hardware and software issues. Hence, the engineer has to master both set of guidelines.

Example 14.2. Control the unknown states

Let us consider the state graph of *Figure 14.6 a)*. Any design requires at least 3 internal Boolean variables to code the 6 states; for instance, *Figure 14.6 b)* shows such a 3-internal variable coding. As a consequence, the sequential circuit has 8 theoretical internal states (from 000 to 111). This may lead to some control difficulties. A simple initial reset of the circuit brings it into a non-defined state (000)! If a parasitic aggression induces '1' values (capacitive charges) in the three flip-flops, then the circuit reaches state (111).

This problem is well known, and it has led to rules that the designers must follow to control the unknown states. Here, we can control the 2

unknown states by adding arcs leading to state 1 supposed to be the initial state of the system (see Figure 14.7).

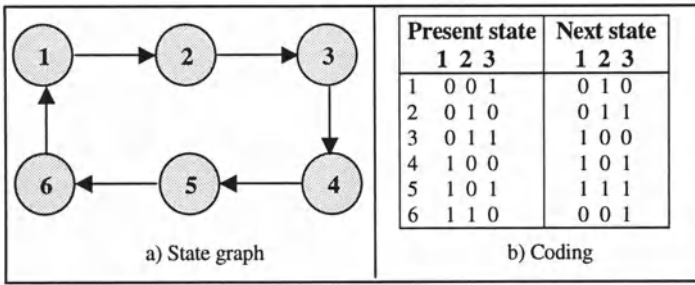


Figure 14.6. State graph coding

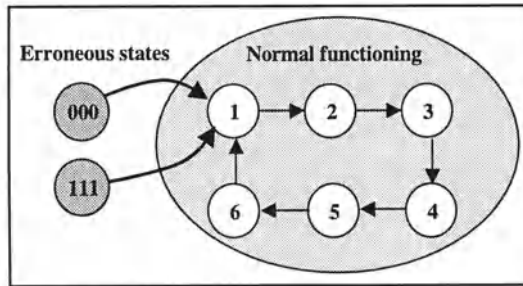


Figure 14.7. Complete state graph

2. Break the feedback loops

As we have already noted in Chapter 12, sequential systems are very difficult to test owing to feedback loops, which reduce the controllability and the observability. Hence, it is important to provide means for ‘breaking’ the feedback loops. This is achieved by the employment of additional signals that allow the blocking of these feedback loops. For example, a sequential circuit thus becomes combinational, which facilitates its test. Figure 14.8 shows an example that uses an AND gate controlled by an external blocking signal. We can thus operate in a step-by-step mode.

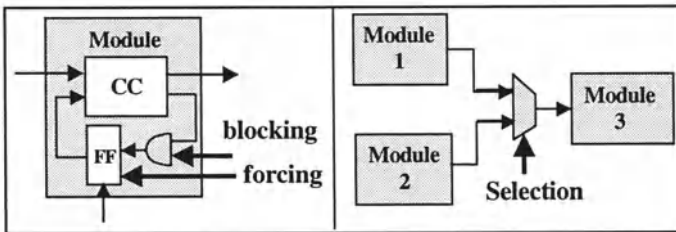


Figure 14.8. Mastering the sequential parts

We can also want to force certain states in flip-flops by use of forcing lines (e.g. reset of one or several flip-flops as illustrated in *Figure 14.8*).

Finally, the insertion of multiplexing functions allows us to control the links between the modules (illustrated by the second schema in *Figure 14.8*).

The majority of design guidelines forbid the employment of asynchronous sequential modules for which the feedback lines are not controlled by clock signals. In fact, such systems are particularly difficult to verify and to test. They react asynchronously to external events and produce dynamic characteristics according to the parameter time. This makes their analysis delicate, and it is difficult to know their internal state by only observing the inputs and outputs.

Test points

1. Insert internal test access points

The controllability and the observability can be artificially increased thanks to the insertion of internal *test points*. In the case of an electronic board, we can add pins or specific connectors linked to the tester by means of specific measurement instruments. In the case of integrated circuits, we can add internal pads that are used uniquely at the time of an under-pin test of the chip with the help of special probes. These techniques have their equivalence in the software domain. For instance, subprograms can be added whose call provides pieces of information about the program state. To illustrate this mechanism, let us give the simple example of a task that uses a stack through two services, Push and Pop. Let us suppose that the call for the service Push is blocked while the stack is full, and that the call for the service Pop is blocked while the stack is empty. The function `Stack_State` returns one of the three values (Full, Empty, Partially), which specifies the state of the stack, but is not useful to the program designer who implements its application. However, this function provides a means of observing the internal state of the Stack.

2. Insert external test inputs/outputs

The debugging and the testing of systems are facilitated by additional test inputs and outputs only handled during test operation. This technique is used a lot and numerous components (microprocessors for example) or complete systems have 'secret' inputs/outputs which are only accessible to the maintenance person. For example, we can control the speed of execution by suspending the activity of the system in order to withdraw internal information; another example is the 'step by step' mode which facilitates debugging.

In the case of software, this possibility is either offered by the execution resources (run-time executive), or introduced into the applicative program

itself by means of statements which either wait for an acknowledge or which write various data that allows the trace of execution to be deduced. This last technique, called *instrumentation*, will also be treated in section 14.2.2.

These additional pieces of information are used notably for the debugging of applications that use micro-controllers and software assemblers. Specific tools exist for test applications, as for example the electronic logic analyzers and the software debuggers.

Test of redundant parts

The presence of redundant elements in a system is a source of great difficulty for testing. Actually, by definition, *passive redundancy* cannot be transformed into failures and cannot therefore be tested. However, it is vital to test these redundant parts at the time of maintenance operations, so as to avoid the masking phenomenon described in Chapter 13.

As already said, a first source of redundancy comes from a non-optimal design. We will try to remove the redundant components and make clean the structure. An example is given in Exercise 14.2.

The second category of redundancy results from voluntary actions, for example to avoid glitches of the signals of an electronic circuit (see Exercise 14.3), or to allow fault tolerance property. It is then necessary to insert additional resources to access these useful redundant elements during the maintenance operations. For example, if we use a Triplex type of redundant structure (studied in Chapter 18), we must be able to deactivate some of them, so as to test each module individually. Another example comes from exception handling mechanisms in programs. These mechanisms introduced in section 14.2.3 allow handling errors. These redundant mechanisms are therefore never activated in normal functionality (without fault activation). The fact that we introduce statements provoking a direct raising of these exceptions allows us to test the exception handlers.

Conclusion

Figure 14.9 illustrates some of the preceding guidelines used to increase the testability of an electronic product after its design: primary input/output test lines, internal test points (control), reset signals (forcing), selection and blocking signals. Although these rules and techniques, which are applied during or after design, are used a lot in industry, they do not guarantee total testability. They constitute part of the initial approach to the methods studied in sections 14.3, 14.4 and 14.5, which are more precise, and more efficient.

In the two following sub-sections, we are going to illustrate two of the preceding guidelines for software applications:

- the *data recording* performed by instrumentation, which improves observability of the past or current execution state of the application (section 14.2.2)
- the *exception mechanisms* offered by some programming languages, whose error propagation mechanism facilitates observability of error presence (section 14.2.3).

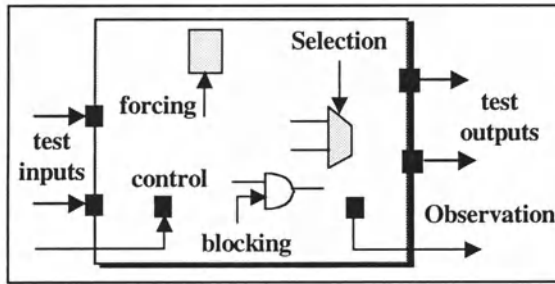


Figure 14.9. Modification of the product

14.2.2 Instrumentation: Data Recording

Instrumentation of software covers two functions:

1. detection of errors occurring during the operation of the application,
2. recording of data characteristic of the execution state of this operation.

The first aspect will be studied in Chapter 16, since it involves on-line error detection means. These means are principally used as the first stage of a fault tolerance mechanism. They can also be used to signal, on-line, the occurrence of an error. The associated fault is therefore diagnosable and correctable after the product has stopped. This is therefore an 'off-line means'. To make the error easy to observe, it is propagated towards the outputs of the system thanks to the exception mechanism. This mechanism will be presented in the next section.

In this section, we analyze the data saving mechanism that instrumentation offers.

The saved data must provide information about the current execution state of the program. They can also store the successive states. This storage of the past is indispensable when we study sequential systems.

This technique is well known among the programmers who display information on the screen that concerns:

- the state of the control flow:

```
printf("Matrix_Inversion starts its execution");,
```

- the state of data flow:

```
printf("Variable Delta contains %i", Delta);.
```

The information saved by the instrumentation mechanism increases the observability of the system, since it provides values on certain internal variables. Instrumentation is thus a mechanism that implements the general guideline '*Insert external test points*' presented in sub-section 14.2.1.

For industrial applications, the data are sometimes saved on disk and more generally in non-volatile memories. This is particularly the case for embedded systems. The volume of stored data must therefore be very limited. Consequently, the engineer has to choose them with care. This data, which characterizes the *Execution State* of the application, have to represent the state of the objects that make up this application as well as the sequencing state of these objects. In the case of a sequential application, the instruction pointer (at machine level) describes this last piece of information, or it is perhaps represented in a more symbolic way. For example, an integer variable can code the number of the subprograms presently executed: its value identifies the branch of the flow of control that is executed. In the case of loops, supplementary information will have to be added in order to store the iteration number.

The information that is explicitly created by the application is contained in the declared variables. It is easily accessible to the programmer. Other pieces of information concerning the state of execution resources also provide useful data for the control of the execution of the system. They must be able to be modified (controllability) or read (observability). For example, these features can concern the internal representation of the data, the management of interrupts, the management mechanisms of time and of tasks (for real-time applications). A language such as Ada offers such features whose standardized description is described in the Ada standard appendices.

14.2.3 Exception Mechanisms: Error Propagation

The observability of the erroneous states occurring at software application run-time is a critical need for the test. For the majority of languages, the executable programs that reach such states continue their execution without signaling these situations. Hence, the error can contaminate the system (error propagation). The resulting failure subsequently makes the diagnosis very difficult. In testing terms, these situations require the addition of vectors that force the propagation of the errors towards the outputs so as to make them observable. To avoid this propagation management from the outside, by the test sequence, certain languages offer a specific feature called the *exception mechanism*, which allows erroneous situations to be handled on-line.

When the handling consists in reaching a safe state and resuming to a normal execution, then the exception mechanism is used to tolerate the fault at the origin of the error (in Chapter 18). If the handling consists in signaling this error to the outside world, the exception mechanism is used to facilitate the test. It is this last situation which is studied in this section and which will be illustrated by using the syntax of the Ada language.

Example 14.3. Exception mechanism

An Ada component is constituted of 3 parts: the *declarative* part, the *body*, and the *exception handler*. Let us consider an example:

```

procedure First is
  ... -- Declarative part
begin
  ... -- Body which contains statements
  ... -- which may raise error E
exception
  when E => ... -- Exception handler
end First;
```

An on-line error detection (see in Chapter 16) occurring during the body execution is signaled by the raising of an exception. In the previous example, E is the exception identifier, that is to say the error name. When such detection is done by the execution resources, a *predefined exception* is raised. For instance, when an index accesses to an array out of its declared range, the predefined exception *Constraint_Error* is raised. When the detection is explicitly programmed by the application designer, the 'raise' statement allows a *user exception* to be raised.

A simple example is given by the following extract:

```

if Passenger_Number > Aircraft_Capacity then
    raise Overbooking;
```

In the previous example, the `First` procedure body contains the statement 'if Condition then raise E;' to signal the occurrence of error E.

When the exception raising occurs, the current control flow is stopped and resumed at the exception handler beginning. For example, if exception E is raised by the `First` body statement execution, the associated exception handler is immediately executed by the statements following 'when E =>'.

If no exception handler exists, or if a raise statement is written in the exception handler, the same exception is re-raised at the subprogram call location. The exception is then *propagated*, in order to take the system structure hierarchy into account (guideline 'hierarchy' of section 14.2.1). This upward propagation mechanism is illustrated by *Figure 14.10*. For instance, if the procedure `Second` calls the procedure `First` which propagated the exception E, then a local handler is searched for exception E

in the procedure *Second* (see *Figure 14.11*). Otherwise, the exception is propagated to the subprogram that calls *Second*, till reaching the main procedure. Thanks to this mechanism, the program user is immediately informed that an error occurred.

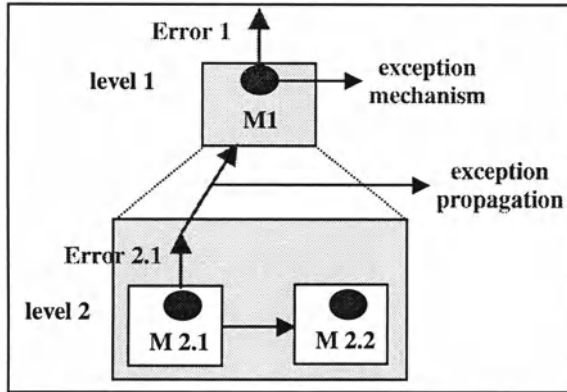


Figure 14.10. Test generation

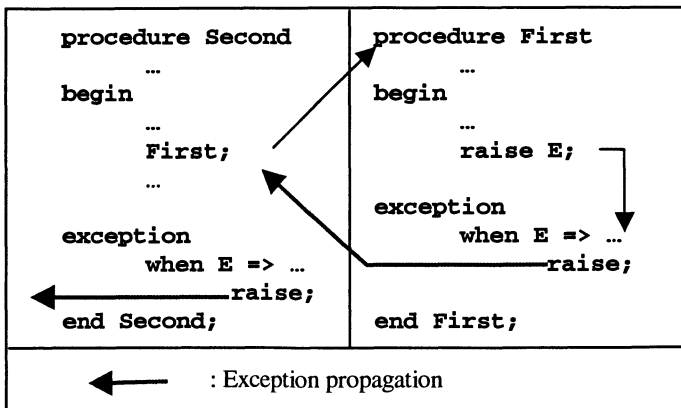


Figure 14.11. Exception propagation

The statements included in the exception handlers may memorize data before the same exception is re-raised. The following program extract illustrates such a situation:

```

exception
  when E => Memorize (Engine_Speed);
  raise E;
end;
```

These data are chosen in order to provide information on the current state (at exception raising time), or the backward propagation path, that is the chain of subprogram calls when the error was reached.

Thanks to the exception mechanism, the errors are propagated up to the output where they can be observed by the tester. The identity of the error, the current state, as well as the propagation path are saved by using the instrumentation mechanism of the previous section, hence facilitating the diagnosis procedure.

14.3 DESIGN OF SYSTEMS HAVING SHORT TEST SEQUENCES

A system can be designed by means of several approaches that end up at final structures which are quite different: electronic components, logic gates, lines of code, tasks, etc. Studies on the testing of logic systems have shown that all of these structures do not have the same potentiality regarding the *testability* of the resulting product. *Specific design methods* that improve the testability will naturally integrate all the remarks made in the previous sections. In this section we want to stress the ‘natural testability’ of some structures, without any addition of extra test points or ports. This concept will be illustrated on hardware and software technologies.

14.3.1 Illustration on Electronic Products

In the case of a combinational logic circuit, the synthesis called ‘linear’ is based upon a mathematical field defined by *Galois* using the XOR and AND operators, without any Inverter. This realization which is ‘canonical’ (only one realization under this form can represent a given logic function) leads to circuits with logic gates which have the remarkable property of being testable with a sequence of fixed test vectors, the length of which is proportional to the number of inputs. Furthermore, we can obtain circuits which are testable by a universal test sequence which is independent of the realized logic function (*Reed-Muller structure* noted RM). Unfortunately, this solution is not always satisfactory with regard to other design criteria, such as the complexity and the response time. Actually, the realization of XOR gates puts a strain on the cost, in terms of number of transistors. Moreover, such easily testable circuits have many logic layers, which makes them slower than traditional circuits.

Now, this approach uniquely has an educational interest. It only has immediate practical interest for calculation circuits (as binary addition is an

XOR function), and coding and decoding circuits used for redundant cyclic codes.

Example 14.4. RM Structure

Let us consider a logic function f of 4 variables expressed by $f = a b' d' + a c' d' + a' b c + b c d$ (x' is the logical complement of x). Let us assume that we realize this function under the form of a two layer SIGMA-PI logic circuit with AND, OR and INVERTER gates, as shown in Figure 14.12. With the classical stuck-at 0/1 fault model hypothesis, the use of structural testing methods (Chapter 13) shows that the testing of this circuit requires the application of 9 test vectors.

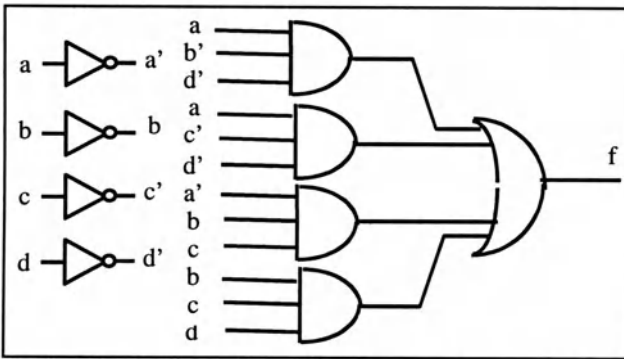


Figure 14.12. SIGMA-PI circuit

Now, let us consider the realization under the Reed-Muller from drawn in Figure 14.13: $f = a \oplus b c \oplus a d$.

If we take the same hypothesis of single stuck-at fault model, the circuit is completely tested by the input sequence of three vectors, $STI(abcd) = (0101, 1010, 1111)$, which gives the respective outputs (0, 1, 1).

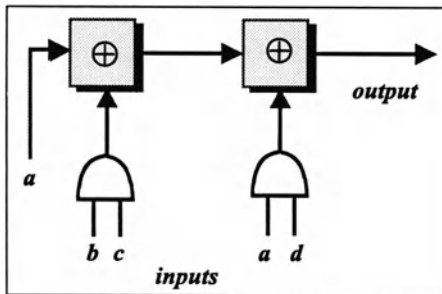


Figure 14.13. RM Circuit

Moreover, if we assume that we must apply all the input configurations in order to test a 2-input XOR gate (4 vectors 00, 01, 10 and 11), then this circuit is totally tested by completing the previous sequence with two other vectors, which gives a sequence of 5 vectors:

$ST_2 = (0101, 1010, 1111, 0111, 1001)$.

The complete analysis of this circuit is conducted in Exercise 14.5.

14.3.2 Illustration on Software Applications

As in electronics, two programs providing the same functionality can have two different degrees of testability. Let us consider the extract of the following program:

```

if (A>B) and (A>C) then Part_1;
                        else Part_2;
end if;

```

If we perform a branch test, it is sufficient to choose values of A and B such that $A \leq B$, in order to execute `Part_2`. Indeed, in this case the local Boolean condition $(A > B)$ is false, thus the global condition ' $(A > B)$ and $(A > C)$ ' is also false. If the compiler has a code optimizer at its disposal, the Boolean expression $(A > C)$ will not be evaluated if $(A > B)$ is false.

Even if it gives an error coverage of 100% for `Part_2`, the validity of this branch test is only partial since the other configuration having to provoke the execution of `Part_2` is not exercised: case $(A < C)$ is false. So that the coverage rate explicitly evaluates the two cases, we would have to write the program in the following way:

```

if (A>B) then
    if (A>C) then Part_1;
                else Part_2;
    end if;
else Part_2;
end if;

```

This program is more complex and less maintainable, since `Part_2` had to be duplicated. On the other hand, it is more testable, since it explicitly distinguishes the two execution conditions of `Part_2`: $(A > B)$ is false or $(A < C)$ is false.

Let us point out that the tests of type Condition/Decision and Modified Condition/Decision, introduced in Chapter 13, respond to this need.

14.4 BUILT-IN TEST (BIT)

14.4.1 Introduction

The *Built-In Test* technique consists in adding to the product a standard specific interface which controls and facilitates access from the external tester, thus increasing the controllability and the observability. As a consequence, the test sequences are simpler, and their application is facilitated. Hence, the tester is also simpler. *Figure 14.14* symbolizes this approach. The drawbacks of the BIT techniques are that they require more components, hence more surface on the chips (increase values of about 15% are announced), and they slow down the signal propagation in the chips. In avionics, the equipment used for maintenance purpose is called *Built-In Test Equipment (BITE)*, cf. ARINC 624 standard.

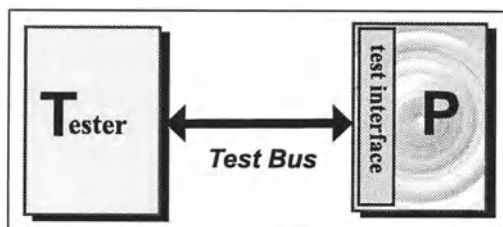


Figure 14.14. Principles of BIT

Several techniques implement the BIT in the electronics domain, such as: the *FIT PLA*, the *Scan Design* which lead to the *LSSD*, the international IEEE standard of the *Boundary Scan* (IEEE standard 1149.1) which is now used systematically to design *ASIC* circuits or microprocessors. We will introduce these techniques in the following sub-sections.

14.4.2 The FIT PLA

14.4.2.1 Structure (Example 14.5)

A Programmable Logic Array (PLA) is an universal logic structure able to implement logic functions in the form of classical SIGMA-PI expressions (as for example $f = a'b + b'c' + b'd + ad' + a'bc$). *Figure 14.15* shows the basic PLA structure comprising an AND net receiving the inputs and elaborating all the *product terms* (such as $a'b$ or $b'c'$), and an OR net elaborating the outputs from the product terms.

The idea of the *FIT PLA* is to modify the traditional structure of a PLA by integrating a parity coding for the product terms (parity noted AND) and

a parity coding for the output functions (parity noted OR). Figure 14.16 shows the general structure of a FIT PLA which implements two logic functions: $f = a'b + b'c' + b'd + ad' + a'bc$, and $g = a'b + b'd + bcd'$.

Each column of cross-points in the AND matrix constitutes an electronic structure realizing an AND logic function. Each point can be realized by a MOS transistor whose gate is controlled by the electric line (associated with the row); all these MOS are connected in series. In the same way, each row of cross-points of the OR matrix represents an electronic MOS structure defining an OR logic function.

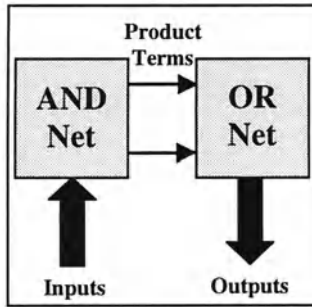


Figure 14.15. PLA basic structure

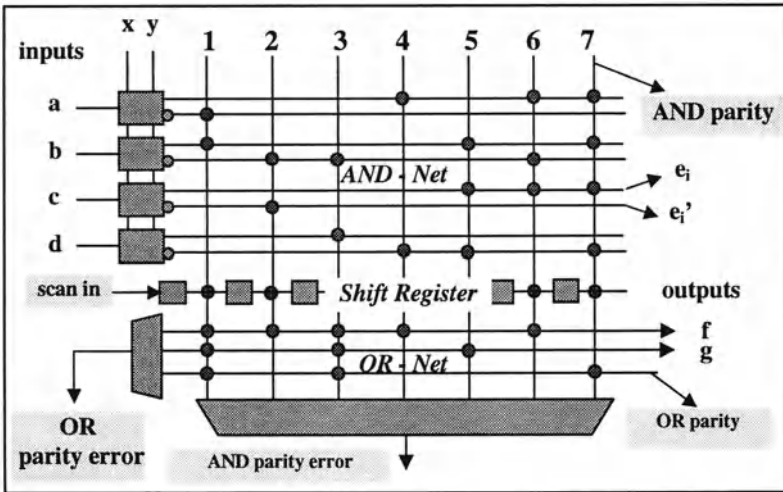


Figure 14.16. Example of a FIT PLA

The aim is to make easier the test of the two matrices (AND and OR), according to faults which affect the interconnections. An additional product term is added which ensures an odd number of 'points' on each line of the AND matrix. This signifies that when a line is at '1', an odd number of MOS

transistors are conducting ('On' state) on an odd number of columns. Similarly, we add an additional line into the OR matrix so that each column of each matrix has an odd number of 'points'. Two parity detector circuits detect if there is an odd or even number of columns in the AND matrix, or of lines in the OR matrix. A last modification is brought to the input amplifiers which receive two inputs, x and y .

The circuit has two different modes of functioning: normal, and test.

- If $xy = 00$, the input circuits create the signals e and e' for a normal functioning of the PLA.
- In 'test' mode:
 - if we apply $xy = 01$, all the rows e_i' are forced to take the value '1',
 - and if we apply the value $xy = 10$, all the rows e_i are forced to take '1'.

14.4.2.2 Test Procedure

The test of the FIT PLA is driven in two stages:

1. test of the AND matrix,
2. test of the OR matrix.

1. Test of the AND matrix. The test of the AND matrix is carried out by initially putting $xy = 01$, and by successively applying the 4 input vectors given in *Figure 14.17*: 0111, 1011, 1101, 1110. Each of these tests is such that only one of the lines of the matrix is at 0, and thus an odd number of columns are at 0. The observation of the output of the parity detector thus detects all the faults that act on this parity. Then, we put $xy = 10$ and we apply the following 4 test vectors <1000, 0100, 0010, 0001>, of which only one bit takes the value 1. This test completes the detection of the previous stage and guarantees the detection of all faults that do not modify the parity (single, triple faults, etc.).

xy	$e_1 e_2 e_3 e_4$	xy	$e_1 e_2 e_3 e_4$
01	$e_i' = 1 \forall i$	10	$e_i = 1 \forall i$
	0111		1000
	1011 4 tests		0100 4 tests
	1101		0010
	1110		0001

Figure 14.17. Test vectors of the AND matrix

2. Test of the OR matrix. The two matrices AND and OR are isolated by a shift register that is able to receive in test mode values from the input Scan In. The test of the OR matrix is based upon the same parity principle as that

of the AND matrix. Across this shift register, we successively apply the vectors (1000000, 0100000.... 0000001), and we observe the parity of the lines with the OR parity checker.

Note. The previous PLA structure which uses an AND matrix and an OR matrix is a symbolic representation of real PLAs. In MOS technology, we use NAND or NOR components. This changes nothing concerning the parity fault detection principle that has been presented.

14.4.3 Scan Design and LSSD

The only way to make a sequential circuit easy to test is to master its sequentiality. In computing systems, circuits are essentially synchronous: their execution is synchronized by a clock signal. Furthermore, the internal state (set of secondary variables) is materialized by a register. An interesting approach, which was imagined at Stanford university in the USA, and gave rise to several methods, is called *Scan Design*. It consists in modifying the state registers, in order to be able to write and read their values from the exterior. Hence, the test of a sequential circuit is essentially reduced to the test of the combinational part, according to the methods already mentioned in Chapter 13 (the *path sensitizing* method, for example).

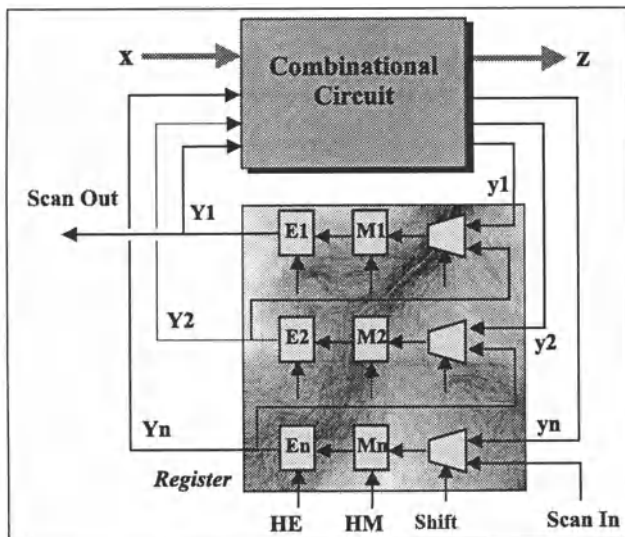


Figure 14.18. LSSD of IBM

The IBM company has popularized the *Scan Design* with the variation called *LSSD (Level Sensitive Scan Design)*, which was used intensively on all large computers since the mythical '360' series of the sixties. This

technique, which is represented in *Figure 14.18*, implies two different functioning modes of the circuit:

- in **normal mode** of functioning, the state register uses ‘Master-Slave’ flip-flops ($M_i - E_i$), and the system evolves to the rhythm of the clocks HM and HE , which are interlaced,
- in **test mode**, this register is transformed into a shift register by setting the *shift* signal at the logic ‘1’. Therefore, it is possible to load the register from the exterior by the input *scan in* with n clock strokes $HM - HE$; this sequence provokes at the same time the reading of the content of the register on the output line *scan out*.

The test of such a circuit is performed by a succession of elementary tests, each of them entailing a cycle of three phases:

1. **SCAN IN:** the signal *shift* is set to the logic ‘1’, and we initialize the internal state of the sequential circuit by n clock pulses $HM-HE$,
2. **NORMAL CYCLE** of functioning: the signal *shift* is set to the logic ‘0’, and we activate an elementary test in the combinational part with one pulse $HM-HE$,
3. **SCAN OUT:** the signal *shift* is set to the logic ‘1’, and we scan out the state register to observe the results of the test by n clock pulses $HM-HE$.

During the steady state, stages 1 and 3 are simultaneous: we load the new state at the same time as we scan out the previous state.

It is unlikely that the tested product is constituted of a single sequential part that is implementing as a single automaton. The *Scan Design* technique is also applied to products that are structured into several interconnected modules. For that, we connect the signals *scan out* and *scan in* of the modules in order to form a chain of ‘series’ tests as shown in *Figure 14.19*.

The test vectors are propagated along this chain as pulse trains. This basic technique is slow, but it can be improved.

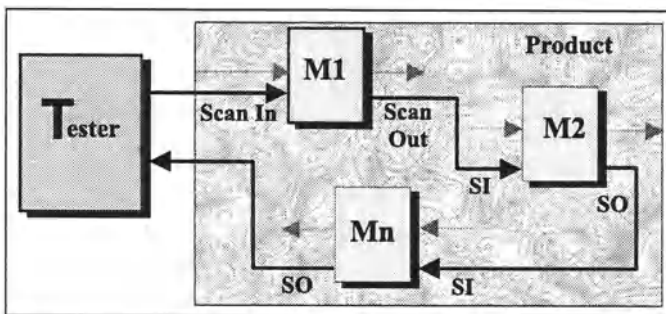


Figure 14.19. Scan Test for Modular Structures

14.4.4 Boundary Scan

Several years ago, a work group of the scientific society IEEE, made up of specialists from industries and universities of several countries, (the *JTAG* - Joint Test Action Group) defined a standardized interface for the testing of integrated circuits: the *IEEE 1149-1* standard. Today, the majority of integrated circuit manufacturers use this standard for ASICs, microprocessors, and micro-controllers (see Appendix C dedicated to dependability techniques associated with a microprocessor).

The *Boundary Scan* ensures the control of the primary inputs and outputs of the circuit according to the *Scan* technique (see *Figure 14.20*), thanks to the addition of some test inputs / outputs and an internal logic:

- The **test bus** (called *Test Access Port, TAP*) composed of specific signals: the series input / output signals, *TDI* and *TDO*, a test clock (*TCK*).
- An integrated logic module comprising:
 - a series register (*Boundary-Scan Register*), for the forcing of test inputs and the reading of the test results,
 - a Bypass Register, in order to reach other modules that are located below in the test chain of the system's modules,
 - an automaton (*TAP Controller*) associated with an instruction register, in order to process certain test operations.

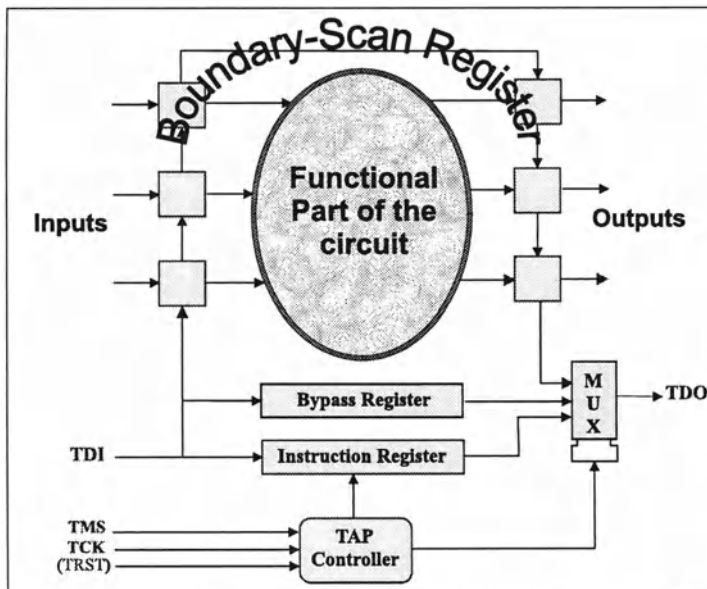


Figure 14.20. IEEE 1149-1 Standard

The automaton (TAP) controls the various operations in normal or test mode. Each cell of the *boundary scan register* has four modes of functioning (see Figure 14.21).

- *Normal mode*: the output multiplexer of the cell transfers the data coming from an input pin of the circuit towards the output of the cell (Data Out), which is connected towards the input of the core logic.
- *Update mode*: the output multiplexer sends the contents of the *parallel output register* towards the output of the cell.
- *Capture mode*: the input data (Data In) is directed by the input multiplexer (Input Mux) towards the *shift register*, in order to be loaded when the *clock DR* occurs.
- *Shift mode*: the bit of each cell of the *shift register* is sent to the following cell via the line *scan out*, whilst the signal *scan in* coming from the previous cell is loaded into the flip-flop of the shift register of the cell.

The output cells are based on the same principle.

These facilities thus allow controlling all the inputs and outputs of the tested circuit, thanks to a shift register that can be loaded and unloaded in series. It is also possible to pass information across a circuit in order to reach a circuit situated below it, or to receive information coming from this circuit.

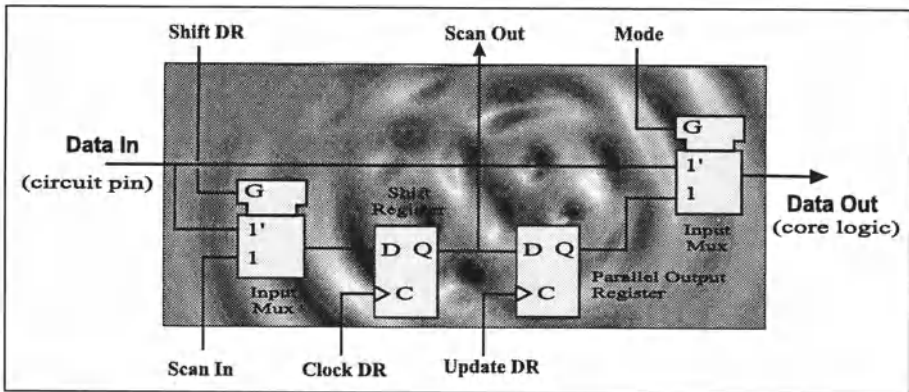


Figure 14.21. Boundary input cell

Notes. In some cases, the boundary scan technique is not applied to the whole circuit, but only to some parts of it. We therefore say that this circuit implements a *partial-scan*, which is opposite to the *full-scan*.

The global use of Scan Techniques in complex circuits, such as those integrated in today communication systems or embedded controls systems, is not always possible. Some parts are really difficult to implement as scan

structures (too much extra-surface or too expensive, too slow, etc.), such as the ROM and RAM memory, which are numerous in today circuits. Some of these parts will be treated separately with BIST techniques (see section 14.5). Scan design is not interesting for circuits having a great number of inputs and outputs, as the big size of the scan register would imply too long test sequences. Hence, these circuits are generally partitioned into several parts, called *scan domains*, which implement separate scan design of about 100 cells. Each one is tested as a unit test, and then an integration test is performed. This allows a good trade off between the test complexity and the controllability/observability mastering. The final integration test of the complete product still remains a real problem.

14.4.5 Discussion about BIT Evolution

As already mentioned, all manufacturers of semiconductors, and in particular the manufacturers of ASIC, have invested enormously in BIT techniques. All present industrial projects of ASIC or full-custom integrated circuits are using the Design For Testability (under full-scan or partial-scan form) in their development platforms. The design costs and silicon surface, which are implied by the scan techniques, are now largely compensated by the advantages brought to the verification of the components (in production and in operation as well). Of course, this progress is made possible by the use of the IEEE 1149-1 standard, which becomes inescapable for technical but also commercial reasons. Other standards favor this development, such as languages and formats for writing test sequences for industrial testers: language BSDL (*Boundary Scan Description Language*), language STIL (*IEEE Standard Test Interface Language*), etc.

Various computing tools for helping in the design and testing have integrated the problems linked to the test of scan type structures: boundary scan support, testability analysis, Design Rule Checking, Automatic Test Pattern Generation, fault simulation, and test vector post-processing. Let us mention two of today prominent companies: Mentor Graphics Corp. and Synopsys. From the descriptions of Verilog or VHDL type, some of these tools automatically produce test sequences in accordance with the STIL format. The BIT testing is generally performed at slow speed, and the stuck-at fault model remains the base of all test operations. With clock rates approaching the gigahertz, other faults must be considered. Thus *timing* or *delay faults* have been added to fault lists treated by most of these tools.

Naturally, the human specialists have still a major role to integrate these separate tools, and to solve the numerous problems not covered by them.

Figure 14.22 shows a typical script of a development based on BIT architecture with one or several scan domains. Functional, structural and test

files coming from previous stages of the development are used by several tools performing DRC, ATPG, and Fault Simulation. The test engineer analyzes the coverage and the possible problems. The solutions to these problems eventually imply to return to the design stage. At the end of this analysis, we obtain a test sequence which can be used by production testing. Such procedure produces today circuits of 10^8 transistors with stuck-at fault coverage greater than 95%.

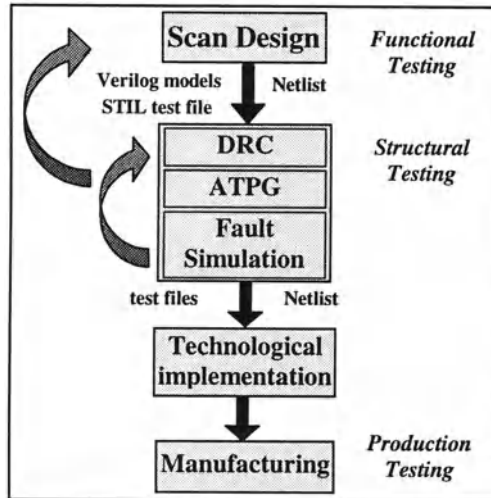


Figure 14.22. Typical BIT script

14.5 BUILT-IN SELF-TEST (BIST)

The test techniques examined in the previous sections require equipment external to the tested circuit: this test is called *off-chip test*. We are now going to tackle another category of test called *Built-In Self-Test (BIST)*, for which the test equipment is totally integrated into the tested product. This is known as *on-chip test* resources. Naturally, this testing improvement needs more components (hence increasing die sizes) and more design investments.

14.5.1 Principles

The BIST approach improves the principle of integration of the tester functions within the product. Of course, such a solution is only acceptable if the complexity and the expenditure of this integrated tester are not excessive. Sometimes, the price to pay for this facility is limited by accepting an important reduction in the coverage of tested faults. This technique allows

the product to test itself, *off-line*, the external tester no longer being necessary. We insist on the ‘off-line’ property of the BIST, as the normal function of the product is suspended during the test. The test operations can be run during power-up phases (for example, test of a RAM by a *galloping* technique presented in Chapter 12), or during maintenance operations. BIST techniques are integrated in more and more industrial products.

The most employed BIST techniques calls upon the test by *signature* described in Chapter 12. Three modules are integrated into the product (see *Figure 14.23*): the *test sequence generator* which is a simple pseudo-random generator, the *compaction function*, and the *signature analysis function*. Naturally, this test cannot detect any failure of the function, as information is lost during the compaction function. An *alias* occurs when an erroneous output sequence provided by the tested function gives the same signature as the fault free signature.

We will develop the pseudo-random BIST techniques in next sub-section.

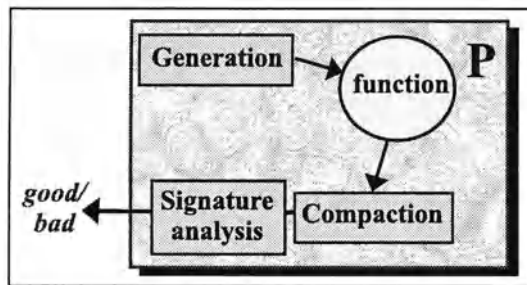


Figure 14.23. BIST principles

14.5.2 Test Sequence Generation and Signature Analysis

14.5.2.1 Pseudo-Random Test Generation with LFSR

A *Linear Feedback Shift Register* (LFSR) is a synchronous sequential circuit, using D Flip-Flops and XOR gates, which generates a pseudo-random output pattern of 0s and 1s.

Example 14.6. 3-bit LFSR

A 3-bit LFSR is shown in *Figure 14.24*. Let us suppose that this circuit has been initialized in state $(Q1, Q2, Q3) = (1, 1, 1)$. Then, the circuit produces a cyclic output sequence with one $(Q1, Q2, Q3)$ output vector for each input clock pulse (see *Table 14.1*). Hence, all binary vectors excepted $(0\ 0\ 0)$ are generated. If necessary, the LFSR basic structure can easily be modified in order to produce the null vector.

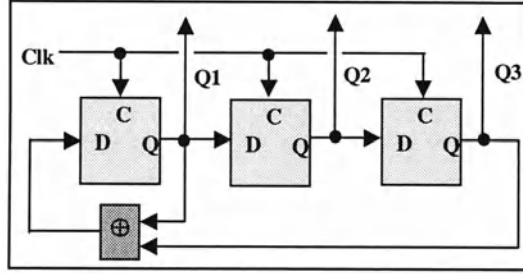


Figure 14.24. 3-bit LFSR example

clock	Q1	Q2	Q3
0	1	1	1
1	0	1	1
2	1	0	1
3	0	1	0
4	0	0	1
5	1	0	0
6	1	1	0

Table 14.1. Produced vectors

Such LFSR can be used as a test sequence generator in an Integrated Circuit as shown in *Figure 14.25-a*. During the test operation, the 3 outputs of the LFSR, *Q1*, *Q2* and *Q3*, are sent to the three inputs of the functional circuit, via a multiplexer.

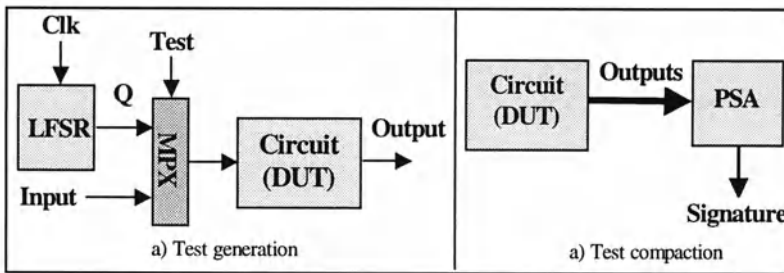


Figure 14.25. Test with LFSR

14.5.2.2 Signature analysis with LFSR

A LFSR can also be used as a compaction circuit, in order to reduce the length of the output sequence coming from the device under test, in the context of BIST testing (see *Figure 14.25-b*). This compaction circuit is called *PSA (Parallel Signal Analyzer)*.

Example 14.7. 3-bit PSA

Let us consider a simple example based on the preceding 3-bit LFSR, as illustrated by the circuit of *Figure 14.26*.

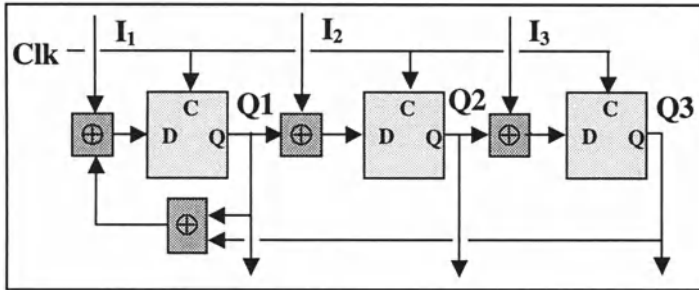


Figure 14.26. PSA with a LFSR

The tested circuit is supposed to produce 3-bit output vectors in response to the input test stimuli (maybe generated by the LFSR of the previous subsection). We suppose here that the PSA is in the initial state (1, 0, 0), and receives one output vector coming from the DUT at each clock pulse.

Input sequence	PSA State Q1 Q2 Q3
Initial state	1 0 0
1 1 1	0 0 1
0 1 1	1 1 1
1 0 1	1 1 0
0 1 0	1 0 1
0 0 1	0 1 1
1 0 0	0 0 1
1 1 0	0 1 0

Table 14.2. PSA response to the input sequence

Table 14.2 shows the evolution of its internal state in response to the given simple 7-vector sequence. If no errors affect the functioning of the circuit, the final PSA signature is (010). Hence, any error modifying this value is detected. This detection capability covers any single bit error, but also a lot of multiple bit errors.

An *alias* occurs when a multiple error in the input vector sequence is masked. Exercise 14.8 analyzes the behavior of this LFSR used as a generator and as a compaction circuit PSA in more detail.

14.6 TOWARDS ON-LINE TESTING

Additionally to the cost and the complexity of the off-line testing, another disadvantage is its 'discontinuous' character: we must wait until the test operation is performed before being able to react on the application in order to correct possible errors or failures. We are now going to show how we can pass from the strict off-line test to an automatic test which executes continuously and is totally integrated into the functioning of the product. Such approach is called *on-line testing* and is described in Chapter 16. On-line testing is necessary when we want to react quicker to errors and failures, in order to obtain more dependability. In this section, we examine three steps:

- placing the tester in the product application site (office, workshop, etc.): this is called *in situ test*,
- facilitating the *maintenance in situ*,
- integrating the test into the normal activity of the product.

14.6.1 To Place the Tester in the Application Site

In a lot of cases, it is more interesting to place the tester into the product site rather than the product into the tester site; this is the case for heavy or fragile products, to increase the intervention speed because of economic consequences of the interruption service, etc. In computing industry, the maintenance operation sometimes amounts to a quick inspection followed by standard exchange of the suspected board, this board possibly being diagnosed and repaired later in a specialized workshop. The maintenance agent travels to the site with his/her test kit, similarly to a doctor who visits patients with his medical kit. The constraint of this approach resides in the necessity to dispose of a 'portable' and efficient tester, and to dispose of access means to the product, thanks to a *test BUS* such as the JTAG BUS.

14.6.2 *In situ* Maintenance Operation

To permit *in situ maintenance*, the product is designed so as to facilitate the diagnosis operations by the maintenance agent. Using again the example of a computer, we will facilitate its test thanks to specialized interfaces, test programs already integrated into the system, etc.

Computer manufacturers also propose a *remote maintenance facility* that allows testing the computer of the client from a specialized center, across a telephone network or through the Internet. This approach involves *in situ* circuitry and software to allow the remote access and diagnosis.

14.6.3 Integration of the Tester to the Product Activity

The last stage consists in integrating the tester into the product and using it in a more or less continuous way, that is during the product functioning. From this point, the proposed techniques deals with *on-line testing*, specified in Chapter 16. Some of these techniques will be of the BIST type. They will be particularly useful when implementing certain fault tolerance mechanisms presented in Chapter 18.

14.7 EXERCISES

Exercise 14.1. Ad Hoc Techniques

Consider a structure of two coupled modules (*Figure 14.27*). Modify this schema by inserting several circuits in order to make the test easier. Explain the expected improvements.

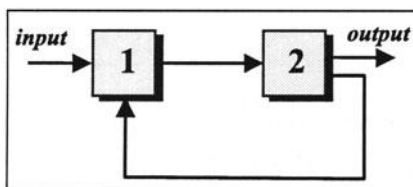


Figure 14.27. Circuit with feedback loop

Exercise 14.2. Analysis of redundant circuits

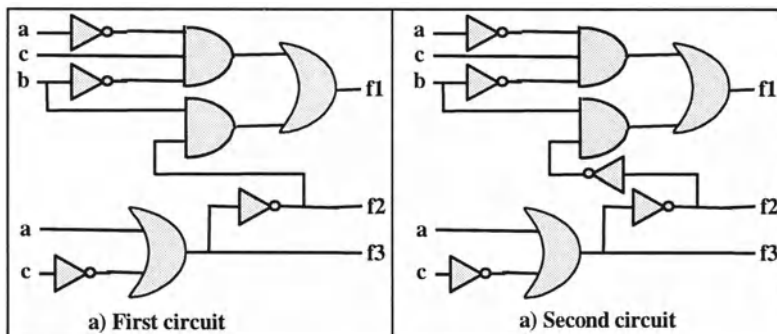


Figure 14.28. Redundant circuits

1. Analyze the gate circuit of *Figure 14.28-a* in order to determine the redundancies; suppress these redundancies and propose a ‘clean’ circuit completely testable.
2. Analyze the circuit of *Figure 14.28-b* and compare it to the previous one. Analyze its testability.

Exercise 14.3. Anti-glitch circuit

The circuit of *Figure 14.29* uses a gate (denoted A) to eliminate the glitches occurring when the input b switches. Unfortunately all stuck-at faults cannot be tested.

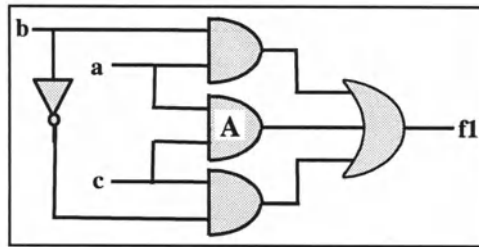


Figure 14.29. Redundant circuit

1. Study the circuit to determine this untestable redundancy.
2. Add a testing input *T* and modify the structure to make it entirely testable.

Exercise 14.4. Easily testable gate network

Logical networks (wired or programmable) constitute very used logical implementation means. Fundamentally they comprise a layer of AND cells which receive primary inputs and their complements, and a layer of OR cells delivering the outputs.

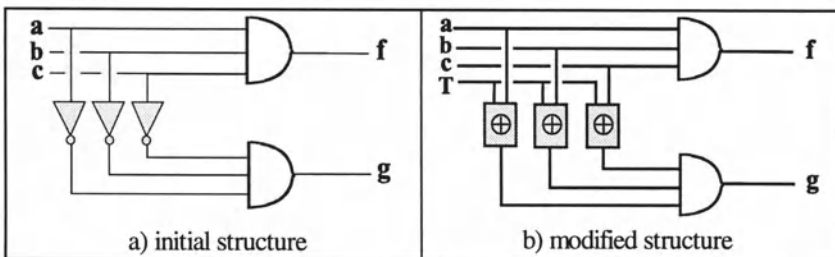


Figure 14.30. Test of a logical structure

Here we are interested in the test of a network of AND gates with a very simple circuit of 3 inputs (a, b, c) and 2 outputs (f and g) drawn in *Figure 14.30-a*. To facilitate the test, we replace the 3 input inverters by 3 XOR gates controlled by a test input noted T : if T has the value '0', then the signals a, b and c are complemented, and if $T = '1'$ they are transmitted without complementation (see *Figure 14.30-b*).

How can this modification improve the test?

Exercise 14.5. Reed-Muller structure

Consider the following logic function of 4 variables:

$$f = a b' d' + a c' d' + a' b c + b c d,$$

and the two circuits implementing this function, as presented in Example 14.4: the SIGMA-PI structure, and the Reed-Muller implementation.

1. We consider the SIGMA-PI realization of this function. Analyze this circuit to find a test sequence as short as possible.
2. Check by inverse transformation (*extraction* of the logic function by analysis of the circuit) that the proposed Reed-Muller circuit realizes the specified function.
3. Check that the test sequence $ST1 = (0101, 1010, 1111)$ covers all the single stuck-at faults of the inputs / outputs of the gates.
4. Specialists of electronic design of XOR functions have proved that to activate all internal faults of such gates, it is necessary to apply all their input vectors. Check that the sequence $TS2 = (0101, 1010, 1111, 0111, 1001)$ satisfies this requirement, and that every internal error propagates to the output where it is observed.

Exercise 14.6. FIT PLA

We want to realize, with the help of a FIT PLA structure (according to the structure described in section 14.4.2), a set of 2 logic functions, $f1$ and $f2$, expressed by the list of their true vertices (noted here $R(i, j, \dots)$):

$$f1(a, b, c) = R(2, 3, 7),$$

$$f2(a, b, c) = R(3, 4, 5, 7).$$

1. Determine the number of *product terms* necessary to implement these two functions in a PLA.
2. Give the symbolic structure of the resulting FIT PLA, by including the parity lines.
3. Find the complete test sequence of this circuit by indicating the faults detected by each vector.

Exercise 14.7. Scan Design

In this exercise, we refer to the technique of LSSD presented in paragraph 14.4.3. We assume that the circuit considered possesses 4 internal variables, and that the combination part is tested with a sequence of 10 test vectors noted $V1$ to $V10$.

Draw a symbolic time diagram that represented the different steps of the test of this circuit, by showing the evolution of the signals HE , HM , $Shift$, $Scan In$ and $Scan Out$.

Exercise 14.8. LFSR

The aim of this exercise is to study the LFSR presented in section 14.5.

1. First of all, it is used as a generator of pseudo-random sequences. Analyze its behavior when its initial state is (111), then (010).
2. The basic LFSR circuit is modified as shown in *Figure 14.31*: the D input of the first D flip-flop receives the XOR of bits $Q2$ and $Q3$. Does this circuit still behave as a LFSR?
3. We will now analyze the compaction circuit of *Figure 14.26* (Example 14.7). Check the compacting sequence given in *Table 14.2*.
4. Still for the PSA circuit, find several non-detectable errors.
5. What do you think about the use of LFSR and of the BIST techniques by signature analysis?

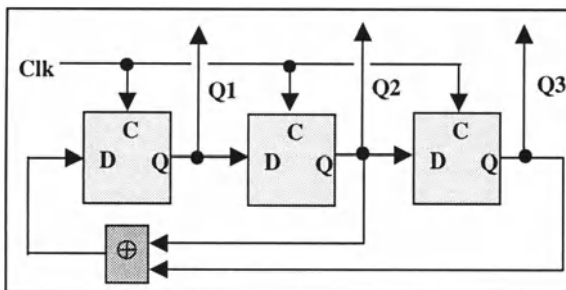


Figure 14.31. Modified LFSR circuit

Chapter 15

Error Detecting and Correcting Codes

In this chapter, *Error Detecting and Correcting Codes* (also noted EDC codes) are presented. These codes, which are an illustration of the general theory of redundancy presented in Chapter 8, were originally used for the encoding of information to allow its transmission in noisy environments; for example, a transmission on an electrical line which is subjected to electrical perturbations. Later on, scientists encountered dependability problems whilst realizing projects for computer systems. They, of course, turned their attention towards the solutions that already existed. In certain cases, it was possible to modify the existing codes, but in many other cases, they had to develop new ways of coding. Finally, more general fault tolerance mechanisms were proposed. Error Detecting and Correcting Codes will be introduced in a very general way. First of all, we will explain the underlying principles, and then we will present the fundamental codes. This information will be very useful, since it will help in the understanding of the following chapters. Actually, combined with other detection techniques that stem from the previously encountered functional redundancy (such as assertions), these redundant codes are employed as a way of detecting errors ‘on-line’. Lastly, they are also useful for fault tolerance, being connected to safeguarding and reconfiguration mechanisms.

15.1 GENERAL CONTEXT

15.1.1 Error Model

We consider a system T , which processes information supplied by the input values U , and provides results at Z (see *Figure 15.1*). This system is

affected by various faults, such as electromagnetic parasites on the electrical lines, or functional faults affecting the function achieved by T . We do not wish to go into details about these faults, so we are going to specify an error model according to the definition given in Chapter 5, that is as *a set of faults characterized as errors by properties on desired or intended states of the system*. In this chapter, the considered attribute that characterizes the behavior will be the output Z of the system. Thus, the desired or intended states are defined by the expected output values of Z .

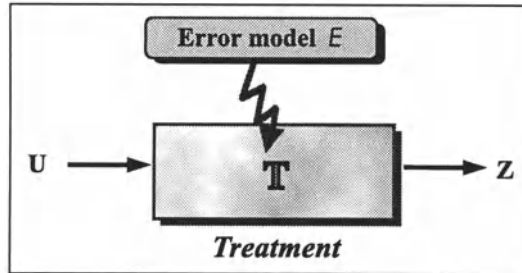


Figure 15.1. Error model

The effect of the faults is represented by a set of *disruptions*, noted E , acting on Z by means of an operator noted '+'. Z^* defines a disrupted output value, i.e. an erroneous state of T . We thus have:

$$Z^* = Z + e, \text{ with } e \in E,$$

The **error model** considered is thus specified by the couple $(E, +)$, where

- E is the set of **disruptions**, which are modifications of correct states causing an error, and
- '+' is the **disruption operator** which combines the correct state with a disruption to express an error.

Note. We have defined errors as erroneous states. In this chapter, we will call error a disruption in order to be in accordance with the conventional vocabulary of *detecting and correcting codes theory*.

These errors are due to the different categories of faults already considered:

- *functional faults* having affected the design or the manufacturing of the product associated to the system T .
- *technological faults* (permanent or temporary) affecting T ,
- *disruptions* due to the environment.

Three error classes are important for transmission and treatment systems:

- an error is *single* when it only affects a single bit of the output Z ,
- an error is *multiple of order p* when it affects at most p bits of Z .

A multiple error of order p is said **error in packet of length l** (also called **burst error**) if the erroneous bits of Z are within an l -distance neighborhood.

These different binary error models originate from *transmission systems*. The input bits U are emitted in series onto a communication media by an emitter. They are received in series by the receiver (and in the same order), and they finally end up at output Z . A single error model is justified if:

1. the parasites which disrupt the transmission are statistically sufficiently distant in time from each other, and
2. their maximum time interval is inferior to the period of transmission of a bit (this basic period is called *moment*).

A multiple error model is used in more aggressive transmission environment. The particular case of errors in packets is confirmed if the parasites have a time duration greater than the basic moment of transmission on the media.

Some examples of errors are given in Example 15.1.

Let us note that the definition of the disruption set (errors) E and the disruption operator '+' must result from an analysis of the product T and from the faults that are able to affect it. In fact, all the detection and correction techniques presented hereafter in this chapter are based on particular error models (such as the single error model for example). Their actual efficiency thus depends on the realism of these models. Therefore, it is generally necessary to carry out an instrumentation experiment on the site, so as to characterize the error model ($E, +$), before all choices are made about codes that detect and correct errors.

Example 15.1. Error Model

We consider a system that serially transmits 4-bit words. We suppose that this transmission is disrupted by electromagnetic parasites that are able to modify a single bit of information. Under these conditions, the error model is defined by the *disruption set* $E = \{1000, 0100, 0010, 0001\}$ and of the *disruption operator* which is a XOR. Hence, if $U = (0\ 1\ 0\ 1)$ and if the *error* (in reality, a disruption) which affects this transmission is $e = (1\ 0\ 0\ 0)$, then the received word affected by the error is:

$$Z^* = (0\ 1\ 0\ 1) \text{ XOR } (1\ 0\ 0\ 0) = (1\ 1\ 0\ 1).$$

This error is a single error since only the first bit has been disrupted. Whatever the transmitted word is, 4 single errors can affect it in this way.

Now, consider the error $e = (1\ 0\ 0\ 1)$. This is a double error, and the same transmitted word $(0\ 1\ 0\ 1)$ becomes $Z^* = (0\ 1\ 0\ 1) \text{ XOR } (1\ 0\ 0\ 1) = (1\ 1\ 0\ 0)$.

15.1.2 Redundant Coding

Error detection and correction require *redundancy*. To illustrate this notion, we consider again Example 15.1. Let U be the value emitted, $Z = U$ the expected value, and Z^* the value received which is affected by an error e . If U can take any 4-bit values, no erroneous output Z^* can be detected or corrected. For example, the reception of (1101) instead of the emitted (0101) cannot be detected as erroneous, as this word is a valid input U .

To be able to detect an error, the system T has to possess a *static redundant functional domain*: the set of values provided by T in the absence of error must strictly be included in the set of values of the output universe. In this way, the output universe is partitioned into 2 subsets: the set of expected values and the set of erroneous values.

Let Z be the original output of T and W a redundant coding of Z . Hence, the universe of W is partitioned into a set of acceptable values called *code* and a set of erroneous values.

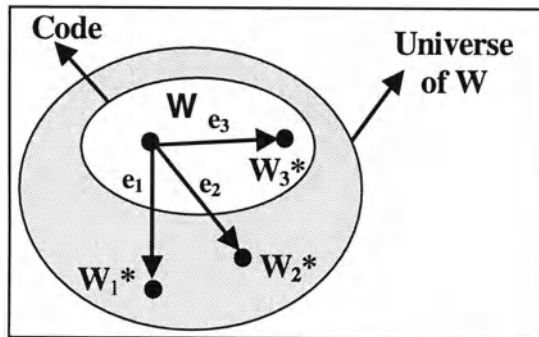


Figure 15.2. Error detection

Let $(E, +)$ be an error model, W a redundant word, and $W^* = W + e$ the word affected by an error e from E . If W^* belongs to the code, the error cannot be detected (case of error e_3 in Figure 15.2). On the contrary, if W^* does not belong to the code, the error can be detected (case of errors e_1 and e_2 in Figure 15.2).

⌊ A *code is an error detector* for the error model $(E, +)$ if and only if $\forall e \in E, W^* = W + e \notin \text{code}$.

To be *error corrector*, a code must allow to find, in addition, the expected value Z from the erroneous value W^* . The following Example 15.2 illustrates these notions.

We will give later on more precise conditions for error detection and correction of a given error model.

Example 15.2. A simple redundant coding

We modify the system of Example 15.1. A fifth bit is added to the output Z to form $W = Z + \{0, 1\}$. We suppose that the value of this fifth bit is '1' if and only if the number of bits of Z having the value '1' is even. The universe of W is constituted of all the 32 combinations of 5 bits, whereas the values of the code form a subset of only 16 configurations. Hence, there is static redundancy. For example, (01101) is a member of the code, whereas (01111) does not belong to the code: its occurrence reveals an error.

15.1.3 Application to Error Detection and Correction

The *Error Detecting and Correcting Codes*, noted EDC/ECC, allow the treatment of errors from a given error model thanks to the use of structural redundancy (see *Figure 15.3*). The inputs U are coded with more bits than necessary (Y) before treatment, then treated by the module T , which delivers a coded result (W), which is then decoded (Z) with the detection and/or correction of possible errors.

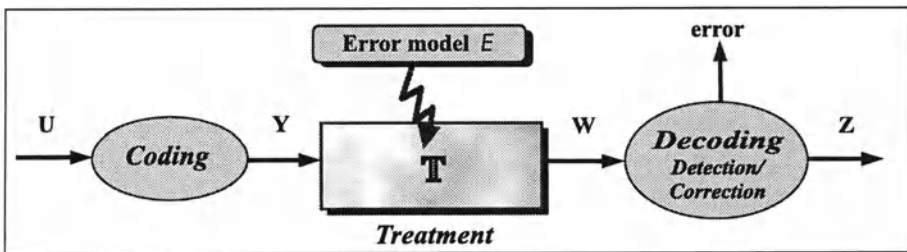


Figure 15.3. EDC code context

If we take Example 15.1 again, error $e = (1\ 0\ 0\ 0)$, which affected the word $(0\ 1\ 0\ 1)$ and produced an output of $(1\ 1\ 0\ 1)$, is detectable if and only if no chosen codeword of output W has the value $(1\ 1\ 0\ 1)$. However, this necessary and sufficient condition gives no information about the way to detect an error, except, of course, by the trivial method of comparison with all words of the code.

So that a code can also be corrective, we must add some constraints in such a way that, after detecting an erroneous word W^* , we are able to discover the correct value W which was affected. Consequently, we are able to deduce from W the correct value of Z . In the following sub-section, Hamming's theorems will provide more detail on error detection and correction. Let us note that, generally, the errors can affect the treatment module T , as well as the coding and decoding parts. These latter parts are functions realized by means of circuits and/or programs, and which therefore

can be altered by errors as well.

Numerous Error Detecting and Correcting codes exist, which are adapted to various situations for various types of data processing systems T : transmission systems (classic codes), memory systems, logic processing systems, arithmetic systems. We are going to give the codes' general principles, describe the most significant codes, and then we will return to the classes of applications in the final section of this chapter. Although the theory of coding is very general and often very formal, we will essentially consider here the *binary codes* by minimizing the theoretical aspects so as to facilitate understanding. However, we will tackle certain codes of decimal type with calculation systems.

15.1.4 Limitations of our Study

15.1.4.1 Anti-Intrusion Codes

We will not develop here the specific category of codes that are intended to protect against *intrusions*. These security codes, which are also applicable to the general context of *Figure 15.1* have the following function: they prevent access to resources with protection keys (*Message Authentication Code*, such as the classic passwords for computer systems access) and to prevent data transformations whilst coding and decoding. The access to world networks, like the Internet, has brought these codes to the attention of the media. Notably, this is because of the regular competition between their implementation and their cracking! Nowadays, two categories of codes are intensively studied:

- the *RSA codes* (Rivest Shamir Adleman), based on the factorization of large numbers, which protect 95% of all electronic exchanges worldwide,
- the *ECC codes* (Elliptic Curve Cryptography) based on the rectification of points on an elliptical curve, such as the standardized IEEE 1363 codes.

15.1.4.2 Low-Level Coding

We also do not consider the *binary-signal* and *signal-binary* coding levels encountered in transmission systems. These levels, close to the transmission media, allow adapting the transmission to throughput problems, signal weakening and noise immunity. They also insure a good synchronization of the transmitted messages. Let us quote as examples three binary-signal coding:

- the *NRZ code* (Non-Return to Zero) which associates electrical levels $+A$ and $-A$ to the '1' and '0' logical values,

- the **Manchester code** which transform any bit '0' into an electrical 0 to 1 transition and any bit '1' into a 1 to 0 transition,
- and the **HDB(n) code** (High Density Bipolar) which transforms any bit '1' into a positive electrical pulse, each bit '0' into a 0 level; a fictitious bit '1' (a pulse) is added after *n* consecutive bits '0', in order to avoid a synchronization loss from the receptors.

These examples are illustrated in *Figure 15.4*. Finally, in many cases of communication media (for example the hertzian one), it is necessary to use adaptation techniques such as the amplitude modulation, the shift keying frequency or phase modulation. These techniques are encountered in the MODEM (Modulation - Demodulation) equipment.

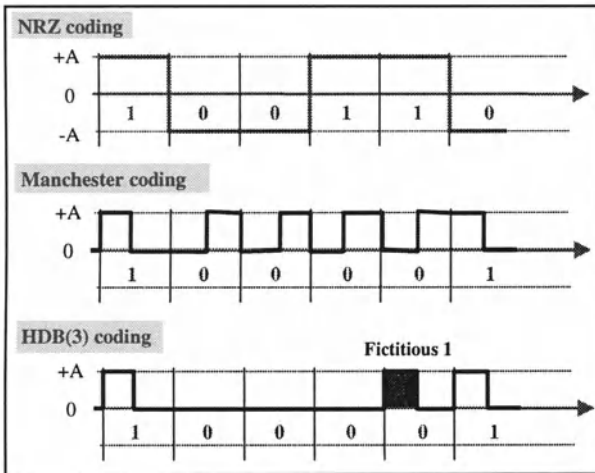


Figure 15.4. Examples of low-level coding

15.2 DEFINITIONS

Let *U* be an information word of *k* bits to code in some binary EDC code, and *Y* the **codeword** (word coded) of *n* bits obtained by the redundant coding: $n > k$.

15.2.1 Separable and Non-Separable Codes

We say that a code is **separable** if all the bits of *U* can be found in *Y*.

Symbolically, we write $Y = (U, R)$, where *R* expresses the *r*-bits **redundancy**, with $n = k + r$. *Figure 15.5* illustrates this notion of separable

code. Let us observe that all k bits of U , which are located in Y , appear grouped in the diagram to simplify the representation: this is not in the least obligatory.

On the other hand, a code is *non-separable* when we cannot directly find U in Y .

The only interest in this property of ‘separability’ resides in the operation of decoding which is much simpler. In fact, the majority of *EDC* codes for transmissions are separable. To decode a non-separable code, we usually have to create a table that contains all the codewords Y and their corresponding initial words U . Except in some particular simple cases, this procedure is highly untractable, due to the size of the table. In numerous applications which do not concern transmission, it is not always necessary to code and decode information at processing time. This allows the use of non-separable codes. For example, the addresses of units communicating via a Bus, or the internal states of a sequential circuit, are objects coded once and for all, at design. If three internal states, states ‘1’, ‘2’, and ‘3’, of a sequential circuit are coded with the words 110, 101 and 011 (corresponding to 3 internal variables, i.e. 3 electronic signals), it will never be necessary to decode these states to find their value (‘1’, ‘2’ or ‘3’).

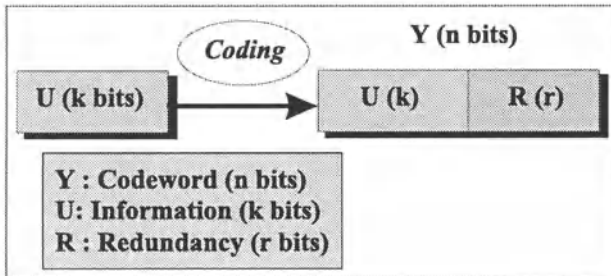


Figure 15.5. Separable codes

15.2.2 Hamming Distance

All *EDC* codes use, either implicitly or explicitly, a concept of *distance* that all codewords must respect: the *Hamming's* geometric distance for transmission codes, and the arithmetic distance for arithmetic codes.

The *Hamming distance* between any two binary words is the number of bits that differ between them.

For example, the distance between $m1 = (1\ 1\ 0\ 0)$ and $m2 = (0\ 1\ 0\ 1)$ is 2. We will express this as $d(m1, m2) = 2$.

A multiple error model of order p applied to any codeword transforms it into a word that is located at a maximum distance p from the correct codeword. We schematize that by saying that the erroneous word belongs to a *sphere* of radius p , centered on the original codeword. The *Hamming's* distance is a *metric distance*:

$$d(m1, m3) \leq d(m1, m2) + d(m2, m3).$$

Hamming stated the properties that a redundant code must possess to be able to detect or correct multiple errors. These properties are based on the minimal distance d that must exist between the two codewords of all the couples $(Y_i$ and $Y_j)$. There are two *Hamming* theorems:

- | A code allows the detection of errors if d is strictly greater to the radius p of the error model: $d(Y_i, Y_j) > p, \forall i \neq j$.
- | A code allows the correction of errors if d is strictly greater than twice the radius p : $d(Y_i, Y_j) > 2p, \forall i \neq j$.

Figure 15.6 symbolizes these properties for error detection. For every couple (Y_i, Y_j) , we have to respect the condition $d > 2p$; every error that affects any codeword Y_i transforms the correct word into an erroneous word Y_i^* belonging to a sphere of radius p around Y_i . Since all the spheres of codewords are disjoint, we can a priori deduce the correct word Y_i from any erroneous word Y_i^* .

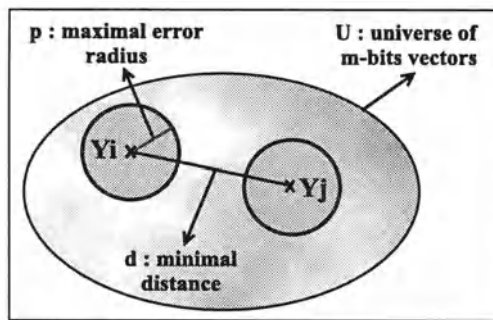


Figure 15.6. Distance and error model

Note. Coming back to the general context of *Figure 15.3*, we see that the detection properties of the codes have to be preserved in the transformation performed by the system T . If T is a transmission system, the normal output W is equal to the coded input Y , and the Hamming distance properties allow the detection and correction of errors. On the other hand, these properties are not automatically preserved when the system T is not a transmission or a memory system. Other notions of distance have then been defined, as we will see with the ‘arithmetic distance’ for arithmetic circuits, in section 15.5.

Example 15.3. Detection and correction of single errors

If the codeword is $Y = (0\ 1\ 1\ 0\ 0)$ and if $p = 1$, all erroneous words are at a distance of 1 from this word. They thus belong to a set of five words:

$$Y^* = \{(1\ 1\ 1\ 0\ 0), ((0\ 0\ 1\ 0\ 0), (0\ 1\ 0\ 0\ 0), (0\ 1\ 1\ 1\ 0), (0\ 1\ 1\ 0\ 1)\}.$$

If none of these words belong to the code, then it is possible to detect the presence of all single errors that have affected the word $(0\ 1\ 1\ 0\ 0)$.

If, additionally, all the other words of the code are situated at a distance which is superior or equal to 3, we will also be able to correct all simple errors. In this way, the word $Y' = (1\ 1\ 0\ 1\ 0)$ is at a distance of 3 from $(0\ 1\ 1\ 0\ 0)$. We can easily check that all simple errors that affect it do not belong to the set Y^* . If this property is true for all codewords, then the detection of an incorrect codeword, which belongs to the set Y^* , clearly identifies the correct word Y : the correction of errors is therefore possible.

15.2.3 Redundancy and Efficiency

Let us suppose again that k is the number of bits of the information to be coded, n the number of bits of the codewords, and $r = n - k$ the number of supplementary bits. A code can be characterized by:

- its *cost*, which is the number of bits n that it needs,
- its *power of expression* (or *cardinality* or even *capacity*), which is the number of codewords N that it is able to represent,
- by the *error model* defining the errors detected and/or corrected.

We use the term *redundancy rate* of the code as the coefficient:

$rr = r/k$, where k is the number of bits of the word to code (called 'useful bits'), and r the number of added bits ('redundant bits').

The *density of a code*, noted d , is the ratio of the *capacity* N and the total number of words that could theoretically be formed with its n bits: $d = N / 2^n$.

The *coverage rate of a code*, noted C , is the ratio of the number of errors that it detects and/or corrects and the number of errors belonging to the considered error model. For example, a *n-bits parity code* detects all odd errors (of order $2.p + 1$) of these n bits. If we consider all possible mathematical errors associated with a given codeword (that is $2^n - 1$), this code covers a large part of them: about $C = 2^{n-1} / (2^n - 1) \approx 0,5!$ Obviously, no real code of finite length can reach coverage of 1.

15.3 PARITY CHECK CODES

The classic codes are the single or multiple parity codes, unidimensional (that is using a single word) or multidimensional (using ‘blocks’ of words). Essentially, they were defined and employed for information transmission systems, and then extended for use in data storage systems.

15.3.1 Single Parity Code

The *single parity code* is the most famous and simplest error detecting code. It is a separable code that adds a redundant bit to U , which is the result of the XOR function of all the other bits:

$$y_j = u_j \text{ for } j \in (1, k), y_n = \bigoplus_{i=1}^k u_i \quad (n = k + 1)$$

The \oplus operator is XOR, which takes the value ‘1’ if and only if the number of inputs that have the value ‘1’ is odd. For example, if $X = (1\ 0\ 1\ 0\ 1\ 1\ 1)$, then we have to add a ‘1’ bit since the number of bits ‘1’ of U is odd. Thus, the codeword contains 8 bits: $Y = (1\ 0\ 1\ 0\ 1\ 1\ 1\ 1)$. Hence, all codewords contain an even number of bits ‘1’. This code does not correct any error, but it detects half of all conceivable mathematical errors, i.e. all the errors that modify the parity of the number of bits ‘1’.

In the previous example, all odd errors (single, triple, quintuple or sextuple error) affecting $Y = (1\ 0\ 1\ 0\ 1\ 1\ 1\ 1)$ are detectable, since they change the parity of the number of bits ‘1’. Finally, the single parity code is the least redundant of all codes that detect and correct errors: $rr = 1/k$; hence, its density is $d = 1/2$.

The single parity code is used everywhere in a computing system: to code data stored in memory, transmitted on a Data or Address Bus, etc. Its limitation comes from the fact that it cannot detect an even number of errors, and in particular double errors (see Exercise 15.1).

15.3.2 Multiple Parity Codes

For *multiple parity codes*, several parity relations (functions \oplus) are defined, and they allow the detection and correction of more complex errors. There are many of these separable codes and they are also very varied.

We are briefly going to introduce *unidimensional codes* that concern the encoding and decoding of individual words, and also *bidimensional codes* that are intended for structured information in blocks of words.

15.3.2.1 Linear Codes

Principles and example

The general principle of *multiple parity codes* is the following: we add redundant bits to the word to be coded; these redundant bits are obtained by XOR relations between certain information bits. These codes were imagined for transmission applications, i.e. for which $W = Y$ without error. We will now explain the basic principles with a simple example.

Example 15.4. Code C(7, 4)

Let us consider the detecting and correcting code, such that $k = 4$ and $n = 7$. The bits y_i of the codeword Y are obtained from the u_i bits of the word to code, U , by the following parity relations:

$$y_j = u_j \text{ for } j \in (1, k = 4),$$

$$y_5 = u_1 \oplus u_2 \oplus u_4,$$

$$y_6 = u_1 \oplus u_3 \oplus u_4,$$

$$y_7 = u_2 \oplus u_3 \oplus u_4.$$

By construction, the words of this separable code are located at distances greater than or equal to three from each other. By applying *Hamming's* first theorem, this code will therefore allow the detection of all errors affecting at most 2 bits. By applying *Hamming's* second theorem, this code allows the correction of all single errors. Explicitly, the detection and correction mechanisms are derived from the construction of the codewords by three parity relations:

$$y_1 \oplus y_2 \oplus y_4 \oplus y_5 = 0,$$

$$y_1 \oplus y_3 \oplus y_4 \oplus y_6 = 0,$$

$$y_2 \oplus y_3 \oplus y_4 \oplus y_7 = 0.$$

These relations are called *parity check relations* or *control relations*. We clearly see that all single or double errors of the transmitted word W affects at least one, two, or all three of these relations. On reception, an error is therefore detectable by calculation and by observation of the *syndrome* which is the 3-bit vector:

$$S = (s_1 = w_1 \oplus w_2 \oplus w_4 \oplus w_5, s_2 = w_1 \oplus w_3 \oplus w_4 \oplus w_6,$$

$$s_3 = w_2 \oplus w_3 \oplus w_4 \oplus w_7).$$

If this vector is different to zero, a single or double error has occurred and it is detected (but it is not correctable). If we assume that the errors are single, this syndrome vector enables the correction of the error. Thus, if the

error affects the bit y_1 , the syndrome has the value $S = (1\ 1\ 0)$, if the error affects the bit y_4 , $w_4 = y'_4$, the syndrome has the value $S = (1\ 1\ 1)$, and so on. This code is called *Hamming code* $C(7, 4)$. In Exercise 15.2 we will study a variant of the presentation, which is obtained by simple permutation of the coding relations; this variation is a more convenient way of correcting errors. In this exercise, we will also introduce the so-called *modified Hamming code*, which detects all single and double errors, and corrects all single errors (without any confusion between these two classes of errors).

Matrix representation

The *EDC* codes based on control relations with XOR operators, have given rise to a large number of studies. An original group of codes, called *linear codes*, uses matrices in the binary *Galois Field* $GF(2)$ (using operators XOR -modulo 2 addition- and AND). Two binary vectors represent the word to code U and the codeword Y . The coding is carried out by a ‘multiplication’ of U by a *generator matrix* called G : $Y = U \cdot G$. A noteworthy property of this linear space is that any linear combination of codewords produces another codeword.

Detection and correction of errors are achieved thanks to a *control matrix* H ; we define the syndrome vector by $S = H \cdot W^T$ (where T stands for transpose). The matrix H can be derived from the matrix G : $H \cdot G^T = 0$. This property expresses the fact that the two vector spaces generated by these matrices are orthogonal.

The syndrome is thus a null vector in the absence of an error, hence:

$$H \cdot W^T = 0.$$

For our example, the two matrices and their properties are:

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}, \text{ with } [y_1, y_2, y_3, y_4, y_5, y_6, y_7] = [u_1, u_2, u_3, u_4] \cdot G.$$

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}, \text{ with } H \cdot \begin{bmatrix} w1 \\ w2 \\ w3 \\ w4 \\ w5 \\ w6 \\ w7 \end{bmatrix} = \begin{bmatrix} s1 \\ s2 \\ s3 \end{bmatrix} = S,$$

S is the syndrome vector identifying the erroneous bit.

When the first k columns of G form a $k \times k$ identity matrix (i.e. containing '1' values only in the diagonal), the code is said to be *systematic*. It is the case of our example. In that case, $Y = [u_1, u_2, u_3, u_4, u_1, p_1, p_2, p_3]$, where p_i are the *parity check bits*.

We will deepen the study of a linear Hamming code in Exercise 15.3.

15.3.2.2 Cyclic Codes

Principles

Cyclic Redundancy Check codes (noted CRC) are derived from *linear codes* by adding the property:

any cyclic shift of a codeword is still a codeword.

Their study can be efficiently done with the help of the polynomial modeling over the *Galois Field* $GF(2)$ using operators AND (noted here \cdot) and XOR (noted here $+$). An n -tuple or vector is expressed as a polynomial by x , x being a symbolic representation variable. The vector $m = [m_0, m_1, m_2, m_3]$ is written:

$$u(x) = m_0 \cdot 1 + m_1 \cdot x + m_2 \cdot x^2 + m_3 \cdot x^3.$$

For example, $m = (1011)$ is written: $1 + x^2 + x^3$.

The generating matrix G becomes an $(n-k)$ degree *generator polynomial* $g(x)$, and the control matrix H becomes the *control polynomial* $h(x)$. A fundamental concept of cyclic codes is that of 'polynomials equivalence modulo a given polynomial'. The most important modulo base polynomial is $x^n + 1$. Two polynomials $a(x)$ and $b(x)$ are equivalent modulo $(x^n + 1)$ if they have the same remainder in the Euclidean division by $(x^n + 1)$. Hence, we write: $a(x) = b(x) \pmod{x^n + 1}$.

The error detection and correction properties of cyclic codes are obtained by choosing generator polynomials from the factorization of the base $x^n + 1$ into irreducible polynomials. *Table 15.1* gives the factorization of $x^n + 1$ for $n = 7$ and $n = 15$:

n	factorization
7	$(1 + x)(1 + x + x^3)(1 + x^2 + x^3)$
15	$(1 + x)(1 + x + x^2)(1 + x + x^2 + x^3 + x^4)(1 + x + x^4)(1 + x^3 + x^4)$

Table 15.1. Polynomial factorization

Any monic divisor of $x^n + 1$ is the generator of a cyclic space. For example, if $n = 7$, $(1 + x)$, $(1 + x + x^3)$, $(1 + x^2 + x^3)$, $(1 + x)(1 + x + x^3)$, etc., are generators of cyclic codes.

In that case, the control polynomial $g(x)$ which generates the orthogonal subspace is easily obtained from the factorization of $x^n + 1$:

$$g(x) \cdot h(x) = 0 \pmod{x^n + 1}.$$

Hence, these two polynomials generate supplementary cyclic spaces.

For example, for $n = 7$, if we choose $g(x) = 1 + x + x^3$, then:

$$h(x) = (1 + x)(1 + x^2 + x^3) = 1 + x + x^2 + x^4.$$

Let $g(x)$ of degree $n-k$ be the generator of a cyclic code. Each codeword will have n bits and will be represented by a polynomial of degree $n-1$, multiple of $g(x)$: $y(x) = a(x) \cdot g(x)$ (with degree of $a(x) \leq n-1$).

From this, a generator matrix can be defined by $G = \begin{bmatrix} g(x) \\ x^1 g(x) \\ x^2 g(x) \\ \dots \\ x^{k-1} g(x) \end{bmatrix}$.

For example, if $g(x) = 1 + x + x^3$, $G = \begin{bmatrix} 1101000 \\ 0110100 \\ 0011010 \\ 0001101 \end{bmatrix}$, the different

columns of G correspond to $1, x, x^2, x^3, \dots, x^6$, from left to right.

In that case, the encoding of a word $u = (1\ 0\ 1\ 1)$ gives the codeword $u \cdot G = (1\ 1\ 1\ 1\ 1\ 1)$.

The control matrix can be derived from the polynomial $h(x)$ as:

$$G = \begin{bmatrix} h(x) \\ x^1 h(x) \\ \dots \\ x^{n-k-1} h(x) \end{bmatrix}.$$

The columns of H correspond to $1, x, x^2, x^3, \dots, x^6$, from right to left.

For example, if $g(x) = 1 + x + x^3$, we saw that $h(x) = 1 + x + x^2 + x^4$.

Hence, $H = \begin{bmatrix} 0010111 \\ 0101110 \\ 1011100 \end{bmatrix}$. We can easily verify that $G \cdot H^T = 0$.

Coding procedure

Now, we will propose a very simple way of performing the coding operation of a cyclic code.

We choose, for example, $g(x) = 1 + x + x^3$, as the generator polynomial of a cyclic code. We will define a systematic cyclic code based on the fundamental property that the polynomial associated with the codeword is a multiple of the generator polynomial. Let $u(x)$ of degree k , the word to be coded, g of degree $n-k$ the generator polynomial and y of degree n the resulting codeword. We multiply $u(x)$ by $x^{(n-k)}$, and then we perform an Euclidean division by the generator polynomial. Thus, we have:

$$x^{(n-k)} u(x) = q(x) g(x) + r(x),$$

$$\text{with } r(x) = p_0 + p_1 x + \dots + p_{n-k-1} x^{n-k-1},$$

$$x^{(n-k)} u(x) + r(x) = q(x) g(x).$$

This relation shows that $x^{(n-k)} u(x) + r(x)$ is a multiple of $g(x)$. Consequently, we can take this polynomial as the codeword $y(x)$ associated with $u(x)$: $y(x) = [p_0, \dots, p_{n-k-1}, u_0, u_1, \dots, u_{k-1}]$.

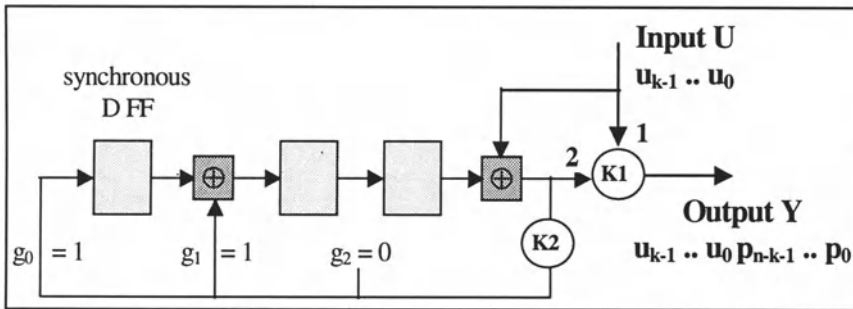


Figure 15.7. Systematic cyclic encoding

This code is systematic. This operation can then be fulfilled by means of a synchronous sequential circuit, which is based on a shift register and XOR operators that feedback some Flip-Flop outputs. For example, Figure 15.7 shows such a circuit for the generator polynomial $g(x) = 1 + x + x^3$. The bits of the vector to be coded are entered into the register, one by one to the rhythm of the clock (which is not represented in the diagram), by starting with the most significant bit u_{k-1} . At the same time, these bits are also transmitted as the most significant bits of the coded word y . During this phase of coding, the switch $K1$ is switched at position 1 (Input U) and the switch $K2$ is closed. We thus perform a series of k shifts which are feedback through the coefficients of the generator polynomial $g(x)$. Therefore, we obtain the $(n-k)$ bits of the remainder in the register. Finally, $K1$ is commuted

at position 2 and $K2$ is open, then we can transmit the contents of the register in $(n-k)$ clock pulses, as least significant bits of the codeword Y . The detailed analysis of this performance is proposed in Exercise 15.4.

The same coding circuit can also be employed for the detection of errors. Actually, an error $e(x)$ affects the word $w(x)$ by transforming it into $w^*(x) = w(x) + e(x)$. It is detectable if the division of $w^*(x)$ by $g(x)$ gives a non-null remainder. We enter the bits of the vector W , by starting with the most significant bit w_{n-1} . When n bits have been shifted, the syndrome $s(x)$ is in the register. In short, the conversion is finished after n clock pulses.

Notes

- *Bose Chauduri* and *Hocquenghem* have proposed a systematic way to construct efficient codes that detect and correct independent multiple errors. These codes, noted *BCH*, belong to the most used CRC codes.
- Certain transmission errors are multiple errors in packets (burst-errors). Thus, specific cyclic codes have been devised in order to detect and correct these types of errors. Hence, *Fire codes* have excellent burst-error correcting capability. They are notably employed in digital disks.
- Cyclic codes are used in very numerous industrial domains. The choice of an appropriate code depends on the nature of the data to be treated, stored or transmitted, and the physical structure that receives and/or treats the data. The determination of a realistic error model is essential. These cyclic codes are generally presented in the documents by their generator polynomial which implies their detection/correction capability. For example, the North America T-carrier standard for transmissions uses the Extended-SuperFrame (ESF) cyclic code for coding frames of 4632 bits; this code which is given by its CRC-6 polynomial, $g(x) = 1 + x + x^6$, is said to detect 98.4% of single or multiple errors. Naturally, such assertion is issued from mathematical analyses and simulation performed to face the error model (1-length burst errors in transmissions, etc.).

15.3.2.3 Bidimensional Codes

We suppose that the information to be coded is structured as blocks, each one constituted of k r -bit words. *Figure 15.8* shows the principle behind the *bidimensional codes*, also called *product codes*.

Two redundancies are introduced:

- rl redundant bits are added to each word, forming what is called *longitudinal redundancy check* (LRC).
- rv redundant words are added to the block, forming what is called *vertical redundancy check* (VRC).

The RR block corresponds to a vertical redundancy of the bits of the horizontal redundancy. Thus, each line or column is a codeword able to detect and/or correct errors.

We can employ linear codes that were studied in the previous section. Exercise 15.5 suggests a study of a bidimensional code with a simple *horizontal parity* on each word and a simple *vertical parity* on all words.

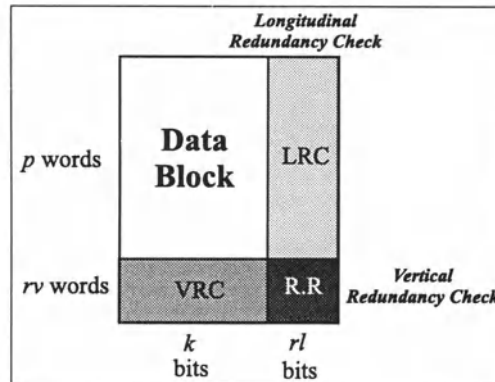


Figure 15.8. Bidimensional code

Note. Bidimensional codes are very useful for optical or magnetic mass storage systems. For example, the *Reed-Solomon* (RS) codes, which are a particular type of BCH codes, are employed in magnetic mass memories, on R-DAT (Digital AudioTape) and digital disks. An interleaving technique is frequently employed: parity symbols are added and the bytes (1,1), (2,1)... are interleaved. Hence we can detect/correct an error in a packet of order n if each word has a single D/C capability! The *Cross-Interleaved Reed-Solomon Code* (CIRC) is used for Audio Compact Disc. Thanks to 2 cyclic codes by block, it allows the correction of short errors in CD manufacturing, and also chained errors. In this way, 4000 consecutive bits can be retrieved, and 12 000 bits can be compensated.

15.4 UNIDIRECTIONAL CODES

The *unidirectional codes* are intended to detect all *unidirectional errors*, i.e. which modify the number of '1' bits in the codeword (either by a greater number or a lower number).

Thus, let m be a word, $m = (1\ 0\ 0\ 1\ 0\ 1)$:

- the errors $m1 = (1\ 0\ 1\ 1\ 1\ 1)$, $m2 = (1\ 0\ 0\ 0\ 0\ 0)$ are unidirectional, as the number of '1' of m (2) is increased in $m1$ and decreased in $m2$,

- the error $m3 = (0\ 0\ 0\ 1\ 1\ 1)$ is not unidirectional, as the first '1' of m became '0', while the 5th bit becomes '1'.

In Appendix A we compare the capacity and the coverage of some of the codes presented below: the simple parity code (which will serve as reference), the optimal m -out-of- n code, the *two-rail* code, the *Berger* code and the modified *Hamming* code.

15.4.1 M-out-of-n Codes

Every codeword of an *m-out-of-n code* has exactly m bits '1' and $n-m$ bits '0'. This code is *non-separable*, so it is used in applications where there is no coding/decoding operation, such as opcode assignment for microprocessors or micro-controllers, or internal state assignment of finite state machines. We will encounter this code in the following chapters, notably to produce self-checking systems. This code detects any error modifying the number of bits '1', in particular any unidirectional error. It has also the following property: any AND or OR combination of two codewords gives a word outside the code. This property finds applications when faults can produce such AND or OR operations, such as the electrical 'wired OR'.

The *power of expression* of this code is the number of different combinations of m '1' that can be formed with a n -bit word: $N = \binom{n}{m}$.

This function has a maximum value when m has an integer value that is close to $n/2$ (see *Figure 15.9*). Therefore, a particular interesting case is that of $n = 2m$, since the cardinality is then maximal. An example of such code is given in Example 15.5.

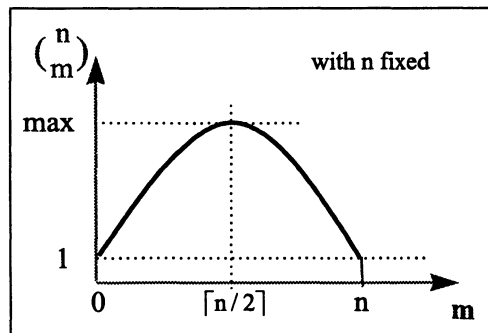


Figure 15.9. m -out-of- n code power with m

The density of the m -out-of- n code is: $\frac{n!}{m!(n-m)! 2^n}$.

Exercise 15.6 deepens the study of the m -out-of- n code by analyzing its properties for the detection of errors.

Example 15.5. Code 2-out-of-4

Figure 15.10 lists the $N = 6$ codewords of the 2-out-of-4 code. It is easy to verify that the minimal distance between two codewords is 2.

	a b	c d	
	0 0	1 1	} double rail (4 words)
	0 1	1 0	
	1 0	0 1	
	1 1	0 0	
} k/n (6 words)	0 1	0 1	
	1 0	1 0	

Figure 15.10. Code 2-out-of-4

15.4.2 Two-Rail Codes

Two-Rail codes (or *double-rail*), which are used to design self-testing logic circuits, coincide with a variant of the *duplication* that has already been encountered. The word to code X is duplicated and complemented to constitute the codeword Y :

$$Y = (X, X').$$

The redundancy R is therefore the binary complement of X , noted X' ($R = X'$). In fact, this is a particular instance of an m -out-of- $2m$ code for the coding of m -bits X words. Its cardinality is $N = 2^m$, which is less than that of a general m -out-of- $2m$ code. Its density is $1/2^m$.

For example, if $m = 2$, only the first 4 vectors of Figure 15.10 belongs to the two-rail code. The two-rail code is thus *separable*, with $k = m$ and $r = m$. Compared to the m -out-of- n code, this code is simpler to implement but less efficient; for a given value of $n = 2m$, we can form fewer words:

$$2^m < \binom{2m}{m}, \text{ for } m > 1.$$

15.4.3 Berger Codes

Berger codes are *separable*: the redundant part R , added to the useful part X , expresses in binary the number of bits '0' present in X . We can show that $r = \lceil \log(k+1) \rceil$, where $\lceil x \rceil$ is the first integer greater than or equal to x .

The density is of the order of $1/(k+1)$.

These codes are optimal codes for the detection of unidirectional errors, i.e. the number of redundant bits is minimal. They have been used for error detection in ALU.

Example 15.6. Berger code with $k = 3$

Table 15.2 gives the list of codewords for the Berger code when $k = 3$: thus $r = 2$ (there must be 2 bits in order to code all the possible ‘zero numbers’ in a word of 3 bits), and $n = k + r = 5$. This example is analyzed in Exercise 15.7.

X a b c	R d e
0 0 0	1 1
0 0 1	1 0
0 1 0	1 0
0 1 1	0 1
1 0 0	1 0
1 0 1	0 1
1 1 0	0 1
1 1 1	0 0

Table 15.2. Codewords of the Berger code, for $k = 3$

15.5 ARITHMETIC CODES

15.5.1 Limitations of the Hamming Distance

Arithmetic codes are specific to calculation systems: addition, subtraction, and sometimes multiplication and division. They are based on the *arithmetic distance* notion, and they are thus efficient to detect *arithmetic* errors. Actually, the *Hamming* distance notion, which is the basis of the majority of codes, is insufficient in the treatment of arithmetic operations. In this paragraph, we therefore take a slight deviation from the binary codes that have been considered thus far.

Let us consider a simple system which calculates the sum of two numbers (see Figure 15.11), $Z = X1 + X2$, and let us quickly examine the problem of detecting errors by the use of redundant codes.

We want to determine a redundant code that detects a certain model of errors affecting the system: let us call $C(X1)$ and $C(X2)$ the two resulting codewords. Having coded the two numbers, we now ask the question:

Is the addition operation an internal operation of the coding?

That is: $C(X1 + X2) = C(X1) + C(X2)$?

A second question concerns the error models of such systems:

What is the physical meaning of a single error model?

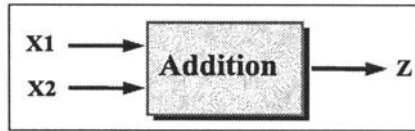


Figure 15.11. An addition function

Let us assume that our addition circuit, depicted in *Figure 15.11*, receives two natural numbers expressed in binary: $X1 = 0001$ and $X2 = 0111$. It provides as result $Z = 1000$. A single error affecting the acquisition of the input $X1 = 0000$ leads to the result $Z = 0111$. We see that a single input error (1 bit affected) produces a quadruple output error (all the bits are affected). Under these conditions, it is not possible to use *Hamming's* theorems, which are based on binary distance! If we want to find codes that detect and/or correct errors altering the addition function T , then we must reject the *Hamming* distance and define a new distance. The notion of **arithmetic distance**, has been proposed (in particular by scientists working in the spatial domain), leading to a new category of codes. In the definition of this arithmetic distance, words are considered as numbers. For example, consider the word $m1 = (1011)$, and two faulty words $m2 = (0011)$, and $m3 = (1001)$. The arithmetic distances between the first word and the two others are: $da(m1, m2) = 8$, $da(m1, m3) = 2$.

Let us note that the two-rail codes that were presented in paragraph 15.4.2 are universal; they can be applied to all data processing systems, and thus calculation circuits. However, they do not optimally exploit the specificity of a system, i.e. its function, as the residual codes, considered in the following paragraph, will do.

Others codes are used. In Exercise 15.11, we propose to study the **checksum code**.

15.5.2 Residual Codes

Here we will only make reference to the **residual codes** which were introduced at the *Jet Propulsion Laboratory* (Pasadena, USA), and were used in the *Saturn V* project. The numbers that T treats are classed according to a 'congruence modulo A ' property, A being a suitable constant called the *check base*. Thus, the class '0' is constituted by the set of numbers $\{0, A, 2A,$

...}, the class '1' by the set of numbers $\{1, A + 1, 2A + 1, \dots\}$, and so on. The congruence is a property that is preserved by addition, subtraction and multiplication. If we perform one of these operations on natural numbers that are expressed in some numeration base B , then we obtain the following property:

$$\text{If } a_1 \equiv a_2, b_1 \equiv b_2 \pmod{A},$$

$$\text{then } a_1 + b_1 \equiv a_2 + b_2 \pmod{A},$$

$$a_1 - b_1 \equiv a_2 - b_2 \pmod{A},$$

$$a_1 \times b_1 \equiv a_2 \times b_2 \pmod{A}.$$

On the other hand, the division does not preserve this property.

This property is exploited in order to check if an operation op (+, - or \times) on a_1 and b_1 is correct. This is achieved by redoing the calculation on the smallest representative of the class (number inclusive between 0 and $A-1$ since there are A classes), and by comparing the class of the result with that of $a_1 op b_1$. This redundant calculation is very simple since it operates on small numbers. Hence, we are able to assume that the circuit that performs this calculation will have a better reliability.

The search for the class of some number N requires the calculation of the remainder of the division of N by A . It has been shown that this calculation is very simple when A is a power of base B minus 1: $A = B^k - 1$. For example: $A = 15 = (2^4 - 1)$ for base 2, $A = 9 = (10 - 1)$ for base 10.

With such values of A , the processing of the remainder of the division by A can be performed by using iteration on k -symbol slices of N , without taking the carry into account.

For example, if $B = 10$ (decimal), and $k = 1$:

$$257 [9] = 2 + 5 + 7 [9] = 14 [9] = 1 + 4 [9] = 5 [9].$$

In this way, we detect all errors (whatever the origin: breakdown, functional fault, external parasite) except errors modifying the result by a multiple of A .

The logical structure of the calculation circuit, which detects the errors in the case of an addition, is shown in *Figure 15.12*. We notice that the redundancy is separable, which greatly facilitates the development and the implementation of the circuit.

An example of the application of error detection within this family of codes is the *modulo 9 proof*, which was well known among previous generations of students; this code is studied in Exercise 15.9. A simple binary application is considered in Exercise 15.10.

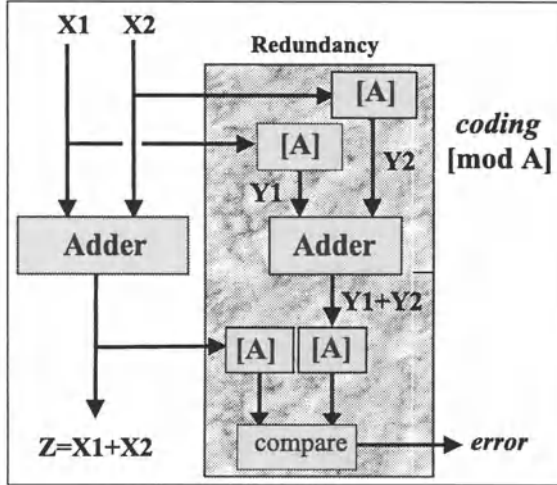


Figure 15.12. Addition circuit that detects errors

15.6 APPLICATION OF EDC CODES TO DIFFERENT CLASSES OF SYSTEMS

The general model of codes considered in this chapter and illustrated in *Figure 15.3*, covers different classes of detecting and correcting codes. We distinguish two cases of applications, according to whether T has the value '1' (a function which represents the typical case of data transmission, since the output is equal to the input), or whether it is different to '1' (the typical case of data processing). Each of these cases then subdivides again into two sub-classes:

- $T = 1$ the output is equal to the input, and so the treatment is either a *transmission* (in this case $W = Y$), or a *data storage* (ROM or RAM) which is, from the functional point of view, equivalent to a transmission;
- $T \neq 1$, the product is performing a *logical treatment*, or an *arithmetic treatment* (which is a very special case of a logical treatment).

When $T = 1$, the classical EDC codes can be efficiently used, as they were designed to handle this case. Consequently, these codes can be used in memory testing. Thus, the *Hamming codes* allow the detection and/or correction of the faults that affect Random Access Memory (RAM) or Read Only Memory (ROM) circuits. Variations of the *Fire codes* are employed to code information stored on magnetic or optical discs.

On the other hand, when $T \neq 1$, these classical codes are generally not suitable. Furthermore, they are not at all adapted for the detection/correction

of functional or hardware faults, or even perturbations that affect the circuits. Various codes are used, such as the single parity codes, or specific codes such as the *m-out-of-n* codes, the *double-rail* codes, the *Berger* code, and the *arithmetic codes* for the calculation circuits.

This chapter performed a unified presentation of codes that detect and correct errors. This approach allowed us to illustrate the use and the interest of redundancy in the detection and correction of faults within two classes of systems: transmission systems and data processing systems. In the following chapters, we will encounter direct applications of these codes, but also other subtler redundancy techniques.

15.7 EXERCISES

Exercise 15.1. Single parity code

The codewords of a parity code are obtained by adding a parity bit, i.e. such that the number of bits '1' in the codeword is even. Consider the case where $m = 4$.

1. Find the codeword of the useful word (1 0 1 1), and determine all detectable errors.
2. Give an example of a non-detectable error.
3. Calculate the following characteristics of this code: *capacity*, *density*, *coverage rate*, and *redundancy rate*.

Exercise 15.2. Hamming Code C(7, 4)

We consider a multiple parity detecting and correcting code such that: $k = 4$ and $n = 7$. The bits of the codeword y are obtained from the word to be coded u in accordance with the following parity relations:

$$y_1 = u_1 \oplus u_2 \oplus u_4,$$

$$y_2 = u_1 \oplus u_3 \oplus u_4,$$

$$y_3 = u_1,$$

$$y_4 = u_2 \oplus u_3 \oplus u_4,$$

$$y_5 = u_2, y_6 = u_3, y_7 = u_4,$$

where u_i and y_i are the bits i of u and y .

1. Analyze this code and show that it detects all single errors **and** all double errors.
2. Show that this code **only** detects and corrects all single errors.
3. The definition of this code corresponds to a simple exchange of the

relations given for the code of Example 15-4. Compare the detecting and correcting capabilities of these two codes.

4. How can we modify this code so that it is able to detect all single and double errors AND correct all single errors (without making any confusion between them)?

Exercise 15.3. Linear code

Reconsider the previous exercise by regarding the *Hamming* code as a linear code.

1. Determine the matrices G and H .
2. Check the vector coding operation.
3. Analyze the error detection and correction with the help of the matrix product $H \cdot W^T$.

Exercise 15.4. Encoding of a cyclic code

Consider the cyclic code generated by the generator polynomial $g(x) = 1 + x + x^3$, and the coding circuit shown in section 15.3.2.2. Study the operation of this circuit for coding the vector $U = (0\ 0\ 1\ 1)$, where the bits are ordered from bit 1 (LSB) to bit 4 (MSB).

Compare with the result obtained by the formal polynomial division of $x^{(n-k)}u(x)$ by $g(x)$.

Exercise 15.5. Single parity bidimensional code

Consider a block of five 4-bits words.

1. Explain how to code this block with a single parity bidimensional code. Give a simple binary example.
2. Determine the classes of single and multiple errors that are detectable.
3. Determine the classes of non-detectable errors. Give a significant example.
4. What can we say about an error that is detected simultaneously in columns 2 and 3 and rows 4 and 5?
5. Study the correctable errors and those that are not correctable.

Exercise 15.6. M-out-of-n code

Show that if $m1$ and $m2$ are 2 words of an *m-out-of-n* code, the following properties are true:

1. ($m1$ OR $m2$) as well as ($m1$ AND $m2$) do not belong to the code,
2. the *Hamming* distance between these two words is included between 2

and $2.k$, and

3. every unidirectional error is detectable.
4. How can an error detection system for such a code be implemented?

Exercise 15.7. Berger code

1. Draw the codeword table of a Berger code with $m = 4$. Is this code optimal?
2. Show that this code allows the detection of every unidirectional error. We will analyze this error detection capability with an error that increases the number of '0' bits, first of all on X , then on R , then on the 2 parts. Then, we will reason with an error that reduces the number of '0'.
3. Consider a code derived from a *Berger* code that requires the calculation of the number of '1' bits in X in order to formulate the redundant part R . This code strongly looks like the *Berger* code. Show that this code does not allow, however, the detection of unidirectional errors.

Exercise 15.8. Unidirectional codes

Find every optimal coding (that have the biggest coding capacity) of $n = 10$ bits for the following codes (we will refer to Appendix A which compares several codes):

1. M-out-of-n,
2. Two-Rail,
3. Berger.

Exercise 15.9. Modulo 9 proof

Study the principle of the *modulo 9 proof* for the addition, multiplication and division operations of decimal numbers (base 10).

1. Show that searching for the class of a number is equivalent to searching for the class of the sum of the classes of each figure of the number. This process is iterative.
2. Use the modulo 9 proof to check if the following operations are correct:
 - $189 + 47 = 236$
 - $189 \times 47 = 8867$
 - $189 - 47 = 144$
 - $189 \div 47 = 97$
3. Show with an example that the modulo 9 proof is false for division?

4. What is the class of non-detectable errors with this code?

Exercise 15.10. Binary residual code

Consider a *binary residual code* ($B = 2$) with $A = 15$.

1. Search for the class of number $N = (101111101111001101)$.
2. Check the operation $(00110010) + (01101110) = (10101100)$.

Exercise 15.11. Checksum code

Consider a block of five 4-bit words: (1101, 0011, 1110, 0110, 0101). We first calculate the sum without the remainder of all these words. The resulting 4-bit word w_6 is then complemented to '2', that is to say we make the following arithmetic operation: $r = 2^4 - w_6$. This word constitutes the redundant word which is added to the others data words. This code is called *Checksum code*.

1. Code this block with the help of *Checksum code*.
2. Show that if no error is present, the 'sum without the remainder' of the 6 preceding words must be equal to '0'.
3. What errors do we detect?
4. What errors are not detectable?

Exercise 15.12. GCR(4B - 5B) code

In this exercise, we will analyze a code called GCR(4B - 5B) which has been used for coding data transmitted on a given media as serial 4-bit words. These words are coded with 5-bit codewords as shown in *Table 15.3*.

The analysis of these codewords reveals that there is no interesting Hamming's distance property. Thus, what was the use of this code?

4-bit X	5-bit Y	4-bit X	5-bit Y
0000	11001	1000	11010
0001	11011	1001	01001
0010	10010	1010	01010
0011	10011	1011	01011
0100	11101	1100	11110
0101	10101	1101	01101
0110	10110	1110	01110
0111	10111	1111	01111

Table 15.3. GCR 4B-5B encoding

Chapter 16

On-Line Testing

In this chapter we examine the techniques allowing the integration of error detection operations into the active life of the product, disturbing in the least possible way the operation of this product.

16.1 TWO APPROACHES OF ON-LINE TESTING

On-line testing (or *OLT*) aims at detecting errors during the product operation. This additional activity must not affect the normal functioning of the tested product. In particular, the operation of the product is not halted during the test operations. This requirement is relaxed by saying that the performance loss of the operation should not be below a certain level of acceptability, which is defined at the specification time.

The main objective of on-line testing is the detection of *errors* appearing during the functioning of the product, so as to alert the outside world. On-line testing is therefore generally not concerned with the localization of the faults at the origin of the detected errors. Hence, the techniques will often be completely distinct from those employed in off-line testing. Moreover, on-line testing is not concerned by correction or recovery of the detected errors. These corrective techniques belong to the fault tolerance examined in Chapter 18. However, most of the fault tolerance solutions use the techniques presented in this chapter in order to detect the occurrence of errors before handling them.

Two different classes of on-line testing techniques are defined:

- the *discontinuous on-line testing*, presented in section 16.2, which exploits the natural temporal redundancies of the product in order to test it discontinuously with a fixed or variable periodicity,

- the *continuous on-line testing* or, *self-testing*, presented in section 16.3, which observes continuously the different functionalities of the product to detect error occurrences.

16.2 DISCONTINUOUS TESTING

The *discontinuous on-line testing* is quite close to off-line testing. We will introduce and illustrate this technique with the example of a control system for a petrochemical industrial process. This distributed system is made up of three regulators that are implemented by specialized circuits, Programmable Logic Array (PLA), Programmable Logic Controllers (PLC), or microprocessors that execute software applications. These regulators are interconnected to each other by means of a local network (Figure 16.1). Each regulator controls a process and the interconnection between the regulators ensures a globally optimized regulation. We are going to examine three possibilities which exploit the temporal redundancies of these regulators, to increase the system testing capability: 1) use of an external tester, 2) test performed by one regulator, and 3) test distributed between the regulators.

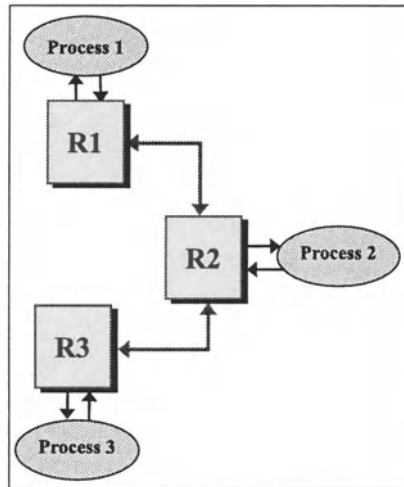


Figure 16.1. Control system

16.2.1 External Tester

Let us assume that the three regulators are not used continuously and therefore they have periods of inactivity. This hypothesis is completely realistic in a lot of cases. On the site, we place a *tester T* connected to the

regulators by a point-to-point network. These new elements are represented in thick black in *Figure 16.2*.

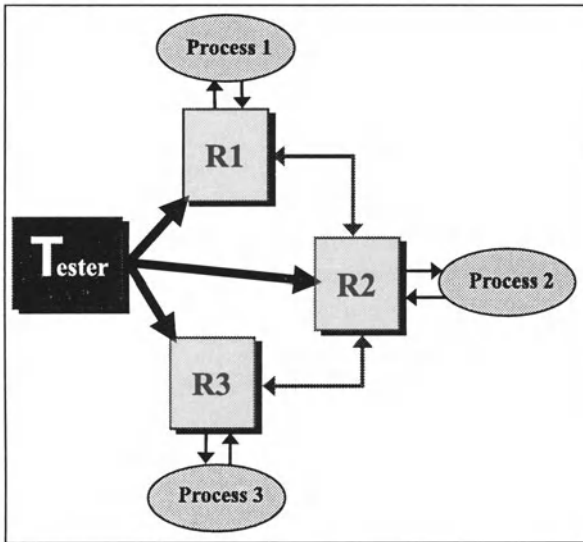


Figure 16.2. Introduction of a tester

The tester alternatively tests each regulator during its inactive period:

- functional testing of the regulation by running processing orders and by comparing the results with predefined values stored in ROM,
- specific testing of certain internal units of the regulator (central unit, arithmetic processor, and memories), and of certain input/output interfaces with the controlled process (analog/digital and digital/analog converters, sample/hold circuits, etc.).

Some modifications of the regulators may be necessary in order to make possible the actions ordered by the tester. For example, when a D/A converter is tested, its output voltage must not be sent to the process, but to an A/D converter instead, to verify these two circuits.

This approach has little practical interest since it is expensive. It requires the installation on the site of a costly tester and communication links with the regulators. An improvement of this schema consists in using the existing local network that links the regulators together. Hence, test data (orders, information) must be interleaved with the communications implied by the regulation functions. The test traffic must not hinder the normal communications. In particular, the traffic due to the test must not be at the origin of failures in the tested systems. Let us suppose that *R1* is waiting for information coming from *R2*. If this information does not arrive before a

certain deadline, then a situation of bad regulation can arise. This failure is not due to a failure of $R2$, but to the fact that the activities of $R2$ are suspended during the test, or because the network is overloaded owing to the exchanges due to the tests.

16.2.2 Test Performed by One of the Regulators

The testing task is performed by one of the regulators which possesses enough calculation power, memory and inactivity periods. In addition to regulating one of the processes, this regulator must test itself and test its colleagues across the natural communication network (*Figure 16.3*). When it can be implemented, this solution is much more economical than the previous one.

The previously highlighted problems of error occurrences due to the test mechanisms are more critical here, since the regulation and the test functions share the same resources: the CPU of the regulator and the communication network.

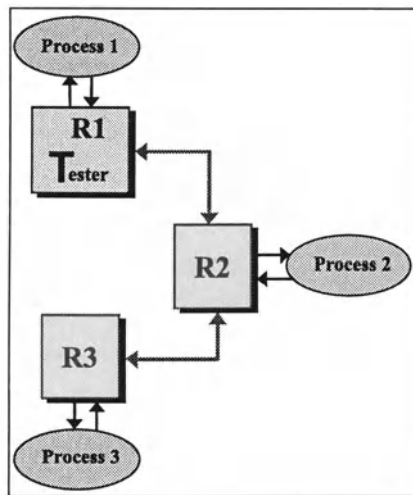


Figure 16.3. Test performed by one regulator

16.2.3 Test Distributed Between the Regulators

With the *distributed test*, there is no longer any centralized test function: each regulator takes care of testing its own operation (*Figure 16.4*) and/or possibly testing its neighbors. This technique assumes that each regulator disposes of enough inactive time in order to activate its test actions, and the necessary hardware and/or software components in order to access to the

data acquisition circuits, the control functions, etc.

Consider the example of a multi-task regulator (see *Figure 16.5*): the activity is scheduled by a fixed sampling period allowing presence of inactive or idle time. We exploit this inactive time to run the test: we add a testing task T having a lower priority.

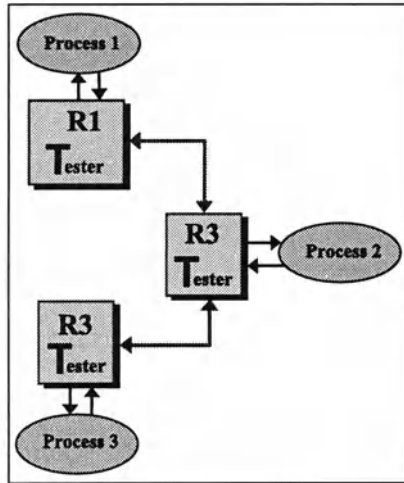


Figure 16.4. Distributed test

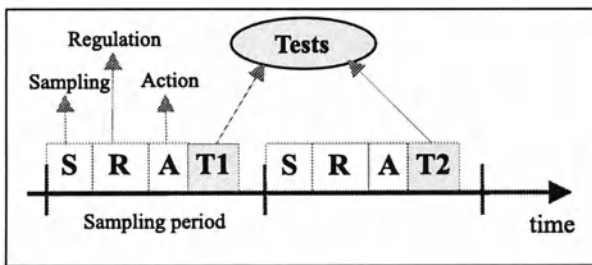


Figure 16.5. Test tasks

Let us note that, unfortunately, the treatment tasks generally have a variable duration, which depends on the calculation functions that have to be executed and on the data sampled. Consequently, the inactive time available at each period is not fixed. If the test task T is not terminated when a real-time interrupt occurs, which starts the next sampling period, this task T is purely and simply cancelled and it must be re-run the next time. Hence, the resulting test frequency of the regulator is reduced.

We noticeably improve this technique by breaking up the test T into n elementary tasks T_i which are integrated into the normal course of

regulation, but with a low level of priority: for each period, we include one or several tasks T_i in a fixed order, for example T_1, T_2, \dots, T_n . This dispatching can be done a priori at the time of design. Another technique, called *periodic server*, adds a periodic task to the application task of which the duration allocated is reserved in order to progress the execution of test treatments. In this case, a periodic test of all the regulators is guaranteed.

Through this example, we explored various *on-line* testing solutions and their property: the various devices are tested with variable or guaranteed periodicity. We have to evaluate these test properties, in order to be able to judge the relevance to the proposed solution faced with the requirements of dependability given in the specifications.

Note. For industrial systems, the on-line test activity is often organized hierarchically. For instance, a distribution control system for electrical power is organized as several units (boards) communicating via a CAN bus. Each board performs the test of its internal memory every 4 minutes: the ROM containing the programs is tested by means of a checksum coding, and the RAM containing the data is tested by 'save - write - read - restore' operations on bytes '00' to 'FF'. A special unit ensures the control of the treatment units by asking each unit to identify itself and to give its internal state every 10 minutes. If a unit does not respond before 500ms, we repeat the request, and in the case of a new failure, the unit is declared as faulty.

16.2.4 Precautions

Whatever the technique employed, the on-line test must not interfere with the functioning which is the concern of the product's operation. In particular, when the test starts, the system is in a certain state which must be preserved at the conclusion of the test. This state concerns: the image that the system possesses of the controlled process (for example, 'the gate is open', 'the contact is off', etc.), the progress of the process control algorithm (for example, 'the calculation of the average temperature has finished').

The absence of disturbances can be guaranteed by saving the *context*, which defines the current state of the system's functioning, at the beginning of the test phase, and then by restoring it at the end of the test. Once the saving of the context has been completed, the functioning state of the product must be placed in an identified and appropriate initial state for the test execution. In the case of a test fragmented into elementary tasks T_i , each task possibly requires a specific initial state. Furthermore, the linking together of these tasks generally requires a saving and restoring mechanism for the *context of the test* (test current state).

The test must also not be at the origin of disruptions of the *resources* that it shares with the application. These resources include the microprocessor

(execution resource), the networks (communication resources), etc. Bad sharing can be at the origin of error occurrence within the application. Such a situation was illustrated at the end of section 16.2.1. The following illustration shows the complexity of the situations that can be encountered. The introduction of saving and restoring mechanisms of the contexts of the application, and also of the test state, must guarantee a non-aggressive functionality between the treatments of the application and those of the test. However, these additional management activities (overhead) consume CPU time and can thus be at the origin of errors, which are due to the sharing of the CPU resource between these two concurrent activities.

The on-line test does not have to disturb the controlled process. Indeed, the actions or outputs of the product, induced by the test, must not be transmitted to the process. This can be obtained by a special device, external or internal according to whether the tester is external or internal. In *test mode*, the product will therefore be partially or totally disconnected from the process, and it is the device that performs the switching or the filtering of the signals. The absence of action on the controlled process at the time of the test does not indicate that no errors may occur. Such an error may be raised, for example, if an actuator is not activated during the test, whereas a periodic refresh operation is indispensable.

Finally, the on-line activity must not raise an erroneous state induced by a wrong test; this is called *false alarm*.

16.3 CONTINUOUS TESTING: SELF-TESTING

16.3.1 Principles

The previous discontinuous on-line test may not be in accordance with the *safety* criteria. Indeed, if we assume that a product has a constant failure rate of λ failures/hour, and that it is tested with a periodicity of Δ hours, then the probability of having a non-detectable failure just before the next

maintenance test operation is: $p = \int_0^{\Delta} e^{-\lambda t} dt$ (cf. *Figure 16.6*). As a first approximation, we can write that $p = \lambda \cdot \Delta$, if λ and Δ are small.

For example, with $\lambda = 10^{-5}$ and $\Delta = 10^3$ hours, we obtain the probability $p = 10^{-2}$ of having a non-detectable fault. This level is high and not acceptable if we compare it with the failure probability 10^{-9} that is frequently required for high-critical systems. Thus, we easily see that if the external consequences of this failure are serious and have a small inertia, then the bigger is the Δ , the greater is the risk generated of occurrence of serious failures due to undetected errors.

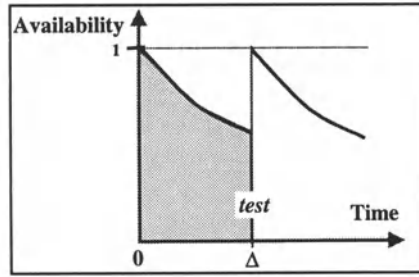


Figure 16.6. Discontinuous testing

The previously described problem comes from long periods of absence of fault detection. Consequently, to improve the dependability of the product in terms of *safety*, we implement more reactive detection techniques that alert the outside world as soon as an error occurs, or better, before an error occurs.

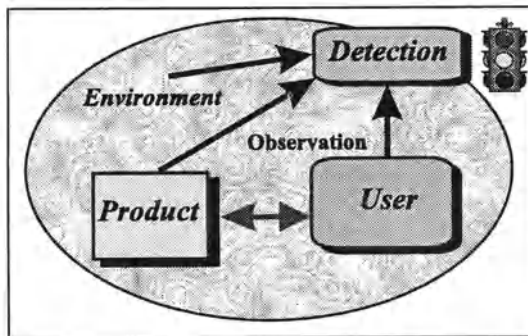


Figure 16.7. Principle of continuous on-line testing

The general framework of self-testing is illustrated by *Figure 16.7*. The detection of errors relies on three types of observation:

1. *Condition monitoring*: observation of the non-functional environment of the product; for example, in an embedded system, we measure the temperature of the environment and/or the voltage distributed by the power regulator; these observations indicate abnormal conditions of operation which are able to lead to faults, etc.
2. *Observation of product operation*. For instance,
 - we analyze, either internally or from the exterior, the response time of a task, of a module, or of the whole product,
 - we carry out a parity test on the data or the address,
 - or we integrate a self-test into the product.

These observations aim at detecting the functioning errors of the product.

3. *Observation of user behavior*: for instance sensor for speed, position or pressure, are introduced into the process controlled by the product; they provide pieces of information about the actual evolution of this process. Their self-test is done by checking that the provided data are within a given range, or by comparing the successive sampled values. These observations also concern a human operator whose behavior is analyzed. For instance, the system asks for useless actions to check his/her availability (dead-man technique).

The detection of non-desirable conditions that occur in the functional and non-functional environment is important for two reasons: anticipating possible faults which are created by side effects on the product, and facilitating the establishment of the symptoms for the diagnosis and recovery after detection.

1. *Prevent possible faults created by the side effects on the product*. For example, the non-respect of the anticipated temperature interval can cause an excessive ageing of electrical circuits. Similarly, bad use of the product, outside of the specifications, can lead the product into a state from which later uses lead to an erroneous state.
2. *Facilitate the establishment of symptoms, first stage for the handling (diagnosis or tolerance) after detection*. If, for example, a functional error of the product is caused by bad use, the establishment of these circumstances is fundamental. For diagnosis purpose, we do not needlessly look for a non-existent fault in the system. For tolerance purpose, it is not necessary to execute an alternative implementation, but the normal use of the product must be recovered.

The techniques presented below are used to detect errors in the fault tolerance approaches. They are often referred to as *self-testing* techniques. However, the reaction to a detected error is not considered in this chapter. For the moment, this reaction is assumed to be external to the couple 'product - process': saving and repairing will be, according to the case, performed by a human operator or automatic hardware or software devices.

The error detection techniques are based on *redundancy* whose two types are successively considered:

- *functional redundancy* by adding checks on behavioral properties,
- *structural redundancy* by modifying the product's structure to allow the detection of errors.

16.3.2 Use of Functional Redundancy

If the product to be tested on-line possesses natural functional redundancies, they are exploited to detect the class of errors that they can reveal. The general notions dealing with functional redundancy were introduced in Chapter 8. Practical examples are given hereafter.

16.3.2.1 Detection Type

The detection mechanisms are integrated into the normal functioning of the tested product, and they observe 'on-line' the truth of predefined properties. The membership to functional domains of certain input, output and/or internal variables are such examples.

We distinguish between three complementary checking types:

- the *pre-condition* which checks whether an operation can be realized before processing it,
- the *post-condition* which analyzes the correction of an operation at the end of its execution,
- the *assertion* which controls if a property is valid each time that a circumstance could lead to violating it.

The *likelihood test*, which consists in checking the membership of a variable to its functional domain, constitutes one of the techniques encountered in data acquisition and treatment systems.

Example 16.1. Pre-condition of a controller

Let us examine a temperature regulation system located in an office. The information provided by the temperature sensor is submitted before treatment, to a *likelihood test* which allows us to detect data capture faults (amplification, signal filtering, analog/digital conversion, transmission, buffer registers, etc.). We check the membership of this calculated temperature to an interval $[Min, Max] = [-50^{\circ}C, +50^{\circ}C]$. Any value out of this range implies an error, which is then signaled.

Let us note that this test is a pre-condition of the temperature treatment function, but it is also a post-condition of the temperature capture function.

Example 16.2. Pre-condition of a subroutine

Consider a piece of software that possesses a subprogram of which an input parameter E is associated with a type defined by an interval $[Min, Max]$. For certain programming languages like Ada, this type implicitly leads to the generation, within the executable code, of test instructions. This

test checks that the value V of the actual parameter E used at the time of the call of this subprogram belongs to this interval: $Min \leq V \leq Max$. In the adverse case, we interrupt the normal running of the program, in order to execute error treatment, or completely stop the subprogram operation by the raising of the exception *Constraint-Error*.

Example 16.3. Assertion

Let us consider the previous example of a program that contains the definition of a type that is constrained by an interval. If a variable is of this type, then every assignment of a new value to this variable generates the verification that this value belongs to the interval.

Unlike the pre and post conditions for which the tests are localized (at the beginning and at the end of the subprogram), assertions are attached to an element (here a variable) and lead to tests each time an action on this element can violate the assertion.

Example 16.4. Post-condition of a subroutine

Consider a subprogram that receives a list of numbers and returns the smallest value (Min) and the largest value (Max). It is evident that the two output variables must satisfy the property: $Min \leq Max$.

Discussion

We can conceive more complex ‘pre’ or ‘post’ condition or assertion properties that the membership to a domain or an inequality between two variables. For instance, two successive values of the same variable are correlated: $V(before) \leq V(after)$. The properties can also concern the flow of control of the execution of the program. For example, the procedure *Initialization* has to be called before the procedure *Treatment*.

The facility to include assertion checking into a product depends especially on the technology employed. In the case of software, we have to insist on the fact that a programming language must be chosen not only according to its power of expression. It should also be chosen according to its capacity to detect errors by static analysis (at compile time) and by on-line testing (at execution time). For example, in the case of the language Ada, the membership expression to an interval is performed very easily:

```
subtype Temperature is integer range -50..+50;
```

The addition of the on-line test also requires design choices, as illustrated by the following example.

Example 16.5

Let us assume that a program contains two procedures P1 and P2. The first one, P1, must be called at least twice before each call to the second one, P2. We have just expressed an assertion on the flow of control (calls to the subprograms). The inclusion of these two procedures in a module will facilitate the implementation of these tests, as shown in the next example where P1 and P2 are placed in a package P of which an extract of the body is provided.

```

package P is
  Number_of_Calls_P1 : Natural := 0;
  procedure P1(...) is
  begin
    Number_of_Calls_P1 := Number_of_Calls_P1 + 1;
    . . .
  end P1;
  procedure P2(...) is
  begin
    if Number_of_Calls_P1 >= 2
      then Number_of_Calls_P1 = 0;
      else raise Error;
    end if; . . .
  end P2;
end P;

```

To conclude, let us signal that the output value of a system or sub-system is not provided before a required deadline. This error can also be detected by a dynamic functional redundancy. The detection mechanism is called *watchdog*.

16.3.2.2 Detection Location

Systems are structures composed of subsystems. An error detection device must be placed in a given place within the structure. This location should be chosen after careful consideration of the characteristics of the errors.

Detection placed within the components

The pre and post conditions specific to the implementation of a component must be placed in this component. They are an integrated part of it, since they must be checked in all circumstances. For example, the operation Pop requires that the Stack is not empty in order to operate correctly. The software implementation is, for example:

```

procedure Pop(E in Element) is

```

```

begin
    if Stack.Empty then raise Stack_Underflow;
    . . .
end Pop;

```

Similarly, a post-condition evaluates the correct functioning of the implementation. For instance, let us consider a procedure that provides the minimum (Min) and maximum (Max) values of a list L. These two results should always satisfy the constraint 'Min <= Max'. Its verification will be made by including a property checking in the body of this procedure:

```

procedure Min_and_Max(L in List; Min, Max out Value) is
begin
    . . .
    if (Min > Max)
        then raise Min_and_Max_Implementation_Error;
    end Min_and_Max;

```

This internal detection does not assume that the fault is internal, i.e. localized within the component. It can also be external, i.e. localized in the surrounding components. Furthermore, the violation of a precondition does not necessarily signal bad use of the component. This error can signal poor implementation of this component. For example, a *Stack_Overflow* error in the *Push* operation of a stack can be caused by a too great a number of stacking (external fault) or by an under-evaluated stack size (internal fault).

Detection on the relations between components.

Errors can be detected as well by checking the relations between the components. It thus expresses an assertion on their cooperation: their sequencing, the data exchanges, etc. The detection mechanism must therefore be placed at the exterior of the components, as it looks at their interactions.

Let us consider again the example treated at the end of the previous subsection. The error detection mechanism implemented in the two procedures P1 and P2 is based on an assertion, "P1 has to be called at least two times before each call to P2", which must be valid whatever the use context of P1 and of P2. This is a property required for a correct functioning of the implementation. On the contrary, if this property is specific to a particular use of P1 and of P2 in a given application, its satisfaction will have to be checked outside of the two components.

As a second example, let us consider two components C1 and C2, whose respective inputs are I1 and I2, and respective outputs are O1 and O2. In the considered application, these two components are executed in sequence; the output O1 of C1 is used as input I2 of C2. We assume that constraints exist on I1 as a result of the behavior of the components situated above C1.

Consequently, the possible values $O1$ produced by $C1$ are themselves constrained. A property P defines the set of these acceptable values. Therefore, the evaluation of this property has to be performed outside of $C1$ and of $C2$, since it concerns their relation. For example, if the components are subprograms, we will write:

```
C1(I1, O1);
  if not P(O1) then raise Error;
                else C2(O1, O2);
                . . .
end if;
```

This example illustrates the error detection on the exchange of values (data flow), whereas the previous example concerned the sequencing of the modules (control flow).

16.3.2.3 Detection Signaling

Let us consider again Example 16.5 presented in sub-section 16.3.2.1. If, when $P2$ is called, the number of previous calls of $P1$ is greater than or equal to 2, then this number is reset to zero so as to count these $P1$ calls again. If not, an error is signaled by using the *exception mechanism*. This example illustrates the fact that, in addition to the detection means, we must also dispose of means that are able to signal the occurrence of an error.

An error parameter could have been used for $P2$. However, this solution does not favor the safety criteria since the subprogram calling $P2$ can be unable to perform the analysis of the parameter after the call (this is a fault which may occur). On the contrary, the raising of an exception ('raise error') will automatically provoke a branching to an exception handler or the propagation to the calling procedure, as described in section 2 of Chapter 14.

Instrumentation

The signaling of an error can provoke the activation of tolerance mechanisms, which aim at avoiding the appearance of a failure. It can also lead to stopping the functioning of the system, or re-initializing this one (hypothesis of transient faults). In both cases, it is useful to implement *instrumentation* procedure in the product: information identifying the error is saved in non-volatile memory (magnetic support, EEPROM), for a future diagnosis of the fault at the origin of the error. Unfortunately, this unique information is most often insufficient to allow a diagnosis in a short time period. For this reason, additional data on the state of the system are also saved: values of input/output parameters, internal variables which characterize the state of the software or of the electronic execution resources (for instance, the state of the internal registers of the microprocessor), etc.

This action, called *fault logging*, stores the error data in a *log file*.

The choice of pertinent data, i.e. facilitating the future diagnosis, is not easy, and it will not be developed in this brief introduction. So as to illustrate this difficulty, we can mention the case of sequential systems, which is the most general case of hardware and software products. The knowledge of the current internal state, at the moment of error detection, is often insufficient. We must therefore save data from the past so as to be able to ‘go back’, from the error up to the fault. Consequently, in this case, it is necessary to save data throughout the operation, even in the absence of errors. The data about the system operation are saved onto a magnetic medium during a time slot of finite duration, so as to limit the size of saved data. In this way, the new stored data erases the older data (notion of instrumentation window).

16.3.3 Use of Structural Redundancy

The use of *structural redundancy* to design self-testing systems is different and complementary to the previous functional approach, and it is also frequently employed in projects of highly-critical systems. This approach involves structural choices at design time, and often calls on, either explicitly or implicitly, the employment of error detecting and correcting codes. We will give first the general definition of a *self-checking system*, then analyze the simple example of the *duplex*, and finally discuss the problem of the test of the *checker*. We will find again EDC codes in Chapter 18, to design fault-tolerant systems.

16.3.3.1 Definition of a Totally Self-Checking System

Let us consider a system with: functional inputs ($x \in X$) and outputs ($z \in Z$), test outputs (w) to which an error detector code ($c \in C$) is associated, and a fault model F .

A *checker*, supposed for the moment to be faultless, observes the w data and raises an error if they do not belong to the code C (*Figure 16.8*).

We define three properties on this system, according to the fault model F :

- *code-preserving*,
- *self-testing*,
- *fault-secure*.

The first property expresses that the fault-free module preserves the output code on w .

The system is said to be *code-preserving* with regard to F if $\forall x \in X \rightarrow w(x, \theta) \in C$,
where $w(x, \theta)$ represents the output w without fault.

The second property expresses that every fault is detectable on the output w by at least one functional input vector.

The system is said to be **self-testing** with respect to F if

$$\forall f \in F \exists x \in X : w(x, f) \notin C$$

The third property guarantees that no incorrect functional outputs can occur which are not immediately detected on w .

The system is said to be **fault-secure** with respect to F if

$$\forall f \in F, \forall x \in X : \text{either } z(x, f) = z(x, \theta),$$

or $z(x, f) \neq z(x, \theta)$ AND $w(x, f) \notin C$

Finally, we obtain the definition of a totally self-checking system:

A **code-preserving system** is said to be **totally self-checking** with respect to F if it is **self-testing** and **fault-secure**.

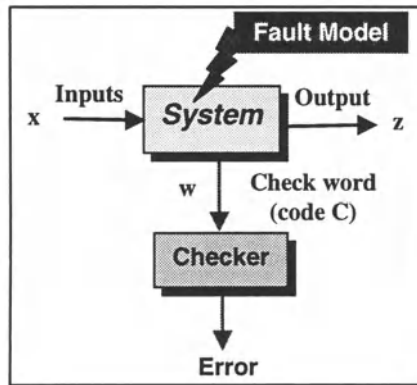


Figure 16.8. Error detection

16.3.3.2 Duplex Example

The most symbolic example of self-testing system is the **duplex**, already introduced in Chapter 8, and illustrated by Figure 16.9: the product is duplicated and the outputs of the two modules are compared to detect possible errors. The figure shows that the test outputs w are constituted of the functional outputs of the main module and of the duplicate (or alternate) module. The error detecting code associated is a duplex code. The checker compares two binary vectors. For electronic systems, it is realized with XOR functions (noted \oplus in the figure).

A variant of this approach, called the **FRC (Functional Redundancy Checking)**, has been proposed by manufacturers of microprocessors like Intel; these microprocessors can be associated by two: a master connected to

the environment, and an observer who checks by duplex the functioning of the first one. Faults detected by this technique comprise all the hardware or functional faults acting on a single block only. Hence, this system is totally self-checking since it is code-preserving, self-testing and fault-secure. Exercise 16.2 and Exercise 16.3 refine the study of the duplex technique.

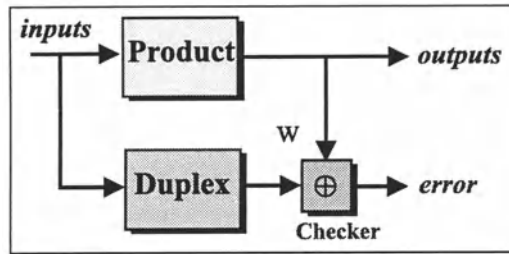


Figure 16.9. Duplex

As soon as the outputs of the duplex and of the normal module are different, an error is signaled (by the error output). Of course, this vision is simplistic. On the one hand, the checker is assumed to be faultless; on the other hand, the real comparison of the outputs of complex products (for example of micro-controllers) has to include a synchronization of the pieces of data which do not occur at the same time. Moreover, the comparison of the two results provided by the modules is not so easy. For instance, if two programs provide numeric results of real type, then the acceptable calculation error has to be taken into account by the checker so as to judge the equivalence of the two results. Furthermore, we cannot accept a simple duplication of the product, since the two duplicates would risk having the same weakness (identical design faults or same sensitivity to perturbations, etc.), and provide the same erroneous outputs, and thus the faults would be undetectable! That is typically the case of the duplication of a piece of software affected by design faults. We must therefore carry out a different design both ‘algorithmically’ and ‘technologically’ for the product on the one hand, and for its duplicate on the other hand. This aspect will be discussed again in Chapter 18, which presents fault tolerance mechanisms.

We have seen in Chapter 15 (dealing with error detecting and correcting codes) that the duplex corresponds to a *separable code* (the *two-rail code*). That is not the only technique used. We can also employ the *m-out-of-n* codes. For instance, we can realize a self-testing sequential circuit from the coding of its internal states with the help of an *m-out-of-n* code. This system therefore possesses the property of outputting the code as soon as a unidirectional error (see Chapter 15) provokes a failure. Furthermore, once it has outputted this code, the system can no longer return to this code.

Note. The term *self-testing* is often used with a completely different meaning to the one given here. Thus, a programmable logic controller is called self-testing by its manufacturer, as it possesses a button and a LED: if we press the button, the LED has to light up if the functioning is tested as correct. Similarly, many ‘self-tests’ are used in many systems when they are switched on: an off-line test program is initiated in order to test certain treatment or memory functions (for example the central memory of a computer). This is certainly not a self-test, but an off-line testing technique of the type BIT, which was studied in Chapter 14.

16.3.3.3 Error Detection Mechanisms: Self-Checking Checkers

In a complex system like a computer, different redundancy techniques for the detection of errors are used in different modules of the structure. Thus, various error detection mechanisms, called here *checkers*, are implemented by circuits or programs, or both, as illustrated in *Figure 16.10*.

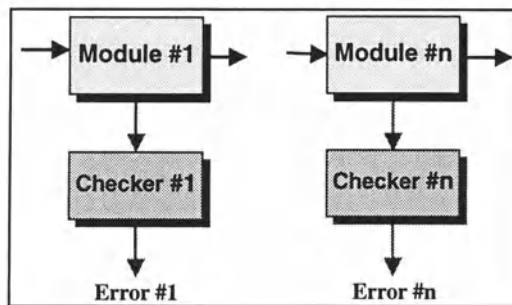


Figure 16.10. Error detection

The different detection functions are rarely independent, since the observed modules are often interconnected. The global management of error detecting mechanisms is therefore complex in many cases. That is true for error detection circuits displaying error signals on a panel, like for example warning lights on the dashboard of a car, or error signaling panels in electro-nuclear power plants. It is also true for software technology, for example as a result of the propagation of exception mechanisms within the different layers of the hierarchy of programs (cf. section 2 of Chapter 14).

Consequently, this additional functionality is also subjected to destructive mechanisms, and all the types of faults envisaged for the modules (functional, technological, aggression faults) can therefore affect it. We are thus led to the problem of testing these redundant parts, either off-line, or on-line. The problem of testing the error observation functions is generally complex. Indeed, they are not directly controllable; thus, the detection of faults altering an error detection circuit may require to artificially provoke

errors of the basic modules.

Independent from the problem of testing detection systems, we must make sure that their complexity is reduced, for dependability and cost reasons. To address these issues and, in particular, to reduce the non-tested on-line parts of the complete product, *self-checking checkers* can be used. We will now briefly explain the principle of these mechanisms.

A *checker* observes a set of n variables belonging to an error detecting code, as for example a parity code, a duplex code, or even an m -out-of- n code. This module produces at its output an error signal as soon as the values of the input variables do not belong to the redundant code, for example the output is a bit which has the value '0' without error and which passes to '1' to signal an error. Very generally speaking, a checker is a *code transformer*. It receives as input I n -bit words belonging to an n -bit input code C_i , and sends to the output O m -bit words (with $m \ll n$) belonging to an output code C_o having at least 2 bits (see *Figure 16.11*).

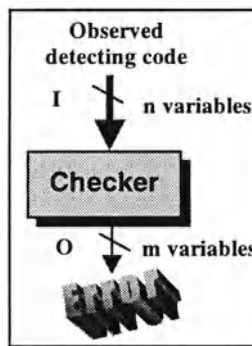


Figure 16.11. Checker

A system is said to be *code disjoint* if:

$$\forall I \in C_i \rightarrow O \in C_o,$$

$$\forall I \notin C_i \rightarrow O \notin C_o.$$

And finally we obtain the final *self-checking* property.

A *checker* is said to be *self-checking* with respect to a defined fault model F if it is *code-disjoint* and *self-testing*:

$$\forall f \in F \exists I \in C_i : O \notin C_o.$$

This property allows us to guarantee that the faults of the checker are detected at the final output when the checked module function correctly. This assumes that all the words of the code C_i are effectively produced by this module. Of course, it is not possible at the output of the checker to know

whether an error is due to a fault affecting the module observed or a fault affecting the checker!

We should note that, even if the tested module does not produce all codewords of the C_i code, the checker could have the self-testing property. Indeed, in many cases, such as the double-rail code or the parity code, a checker can be tested by small sub-sets of these codes. We will analyze this property in Exercise 16.4 and Exercise 16.5.

Finally, if several checkers are used in a system, so as to observe the errors made by different redundant modules, it is sometimes possible to observe the output codes of all these checkers with another checker. This additional checker is a code 'reducer' for the final output code signaling the error (see Figure 16.12). Thanks to this technique, we guarantee that each part of the checkers is tested on-line.

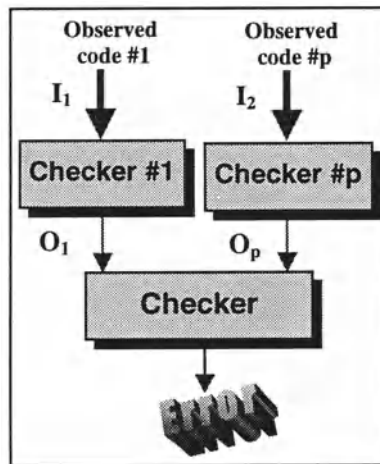


Figure 16.12. Combination of checkers

Thus, the part of the system which cannot be tested on-line (called the *kernel*), is reduced to small circuits or software modules that will be periodically tested off-line. Numerous checker circuits have been proposed. In particular, we can mention the Carter's cells (from the name of a researcher at IBM who proposed them) which uses double-rail codes and which can connect themselves quite easily in the same way as XOR gates, by associativity. A study of SCC (for self-checking checker) for two-rail and parity coding is proposed in Exercise 16.4 and Exercise 16.5.

From these basic principles, many variants have been proposed and used, but they will not be discussed here.

16.4 EXERCISES

Exercise 16.1. Test of a control system

A regulation system is constituted of three interconnected regulators ($R1$, $R2$ and $R3$), each performing the regulation of a unit. A tester is linked to these three regulators in order to apply to them test sequences at the time of periods of inactivity (see *Figure 16.2* and sub-section 16.2.1). We assume that $R1$ is stopped once a week during 1 hour, that $R2$ is at rest each morning from 6H to 6H30', and that $R3$ can be interrupted 10 minutes every hour.

1. Study the work of the tester, its needs in terms of actions on the regulators, and the periodicity of the average test of the equipment, by assuming that the sequences making the complete test of each regulator spend less than 6 minutes.
2. What must we do if the test duration is increased to 15 minutes?
3. Examine the problem of the management of the test of a regulator when the test activity is split into several elementary tasks T_i .

Exercise 16.2. Duplex technique

We consider the diagram of the *duplex* given by *Figure 16.9*.

1. Draw up the inventory of the detected faults and of those that are not detected. In particular, study the influence of the checker on the on-line test property of the product.
2. Comment on the following assertion:

the duplex technique is of no interest, since it divides the reliability of the product by two.

Exercise 16.3. On-line testing of a half-adder

Consider a half-adder which provides the sum and carry functions: $s = a \oplus b$ and $c = a.b$. This circuit was realized by logic gates according to the schema in *Figure 16.13*. We consider the 'stuck-at 0 and 1 of inputs and outputs of gates' fault model.

1. Analyze the existing functional redundancies of this circuit. Deduce the on-line detection capability of this circuit. Discuss the limits of this property.
2. Modify this circuit by adding an output noted p , to create a parity code. In this way, does it detect every single or multiple errors of the complete circuit? Discuss the checker characteristics.
3. Modify the previous circuit by using distinct (independent) logic circuits for each output. What improvements does this therefore bring? What

faults remain undetected on-line?

- Study the same thing with a duplex structure. Analyze every detected single or multiple faults and those that are not detected. Discuss the checker that is necessary for this structure.

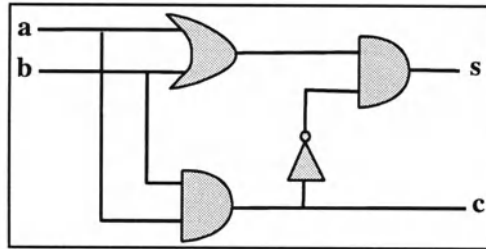


Figure 16.13. Half-adder

Exercise 16.4. Double-rail self-checking checker

A checker receives two groups of inputs a_1, a_2 and b_1, b_2 , and delivers two outputs c_1 and c_2 (cf. Figure 16.14-a). Each pair of signals belongs to the double-rail code, i.e. it is defined by the correct configurations: $\{01, 10\}$. The vectors 00 and 11 thus correspond to error signaling.

We assume a classical single stuck-at fault model.

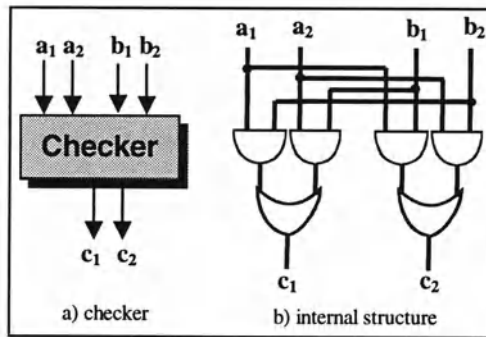


Figure 16.14. Double-Rail self-checking checker

- Show that the logic structure in AND and OR gates proposed in Figure 16.14-b is code-disjoint and self-testing. For the second property, we can use the method studied in Chapter 13 for the search of the faults covered by an input vector.
- Now we put together several of the previous cells in order to treat the double-rail codes of more than two pairs of bits. Study the conditions on the input vectors for which a network of cells has the property of being a

total self-checking checker.

Exercise 16.5. Parity self-checking checker

A checker is intended to detect errors on an input parity code having 4 bits, *a*, *b*, *c* and *d* (3 bits of data plus one parity bit); it delivers two outputs *f* and *g* coded with a 1-out-of-2 code (01 and 10 are the codewords). We realize this checker with two XOR gates as shown in *Figure 16.15*.

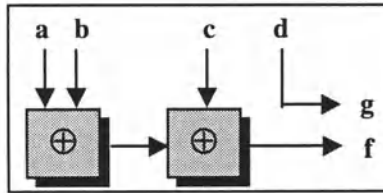


Figure 16.15. Parity checker

1. Is this circuit a self-checking checker? Give an example of a minimal sub-set of vectors of the parity code that ensures the self-testing property.
2. Show that this circuit is no longer self-testing if all the input vectors of the parity code are not applied.
3. Does a permutation of the input variables (*a*, *b*, *c*, *d*) have an influence on the property of the previous question?

Exercise 16.6. Software functional redundancy

We consider a function included in the regulation system of a freezer. This function can only be called when the freezer is in a state ‘freezing’. It receives two temperature values *Min* and *Max*, and returns an intermediate value obtained by an algorithm which is not considered here.

Criticize the following solution in terms of on-line detection of faults.

```

function Intermediate (Min, Max : in integer)
    return integer is
I : integer;
begin
    . . . -- calculation of I
    return I;
end Intermediate;

```

Propose a better version.

Chapter 17

Fail-Safe Systems

In the first part (in Chapter 4), we have mentioned that the failures altering a product can be dispatched into several classes according to the seriousness of their impact on the system itself, on its user, and on its environment. The set of the four classes - benign, significant, serious and catastrophic - constitutes an example of such classification. In this chapter, we consider faults whose external consequences are dangerous, for example serious or catastrophic. Their analysis refers to the *safety* criterion.

With *fail-safe systems*, failures are accepted as long as their external consequences are not dangerous. More precisely, the probability of the occurrence of such non-desired dangerous events must be smaller than a given *acceptance level*. Naturally, such acceptance level depends on the seriousness of the external consequences of the failures. Hence, we introduce the notion of *risk*. For example, the specifications of a critical project express that the risk of catastrophic failures must be smaller than 10^{-9} .

This approach refers to the fault tolerance approach, according to the analysis made in Chapter 6. However, we distinguish these techniques from the more sophisticated fault tolerance techniques presented in Chapter 18 for pedagogical reasons. Here, the product may produce a failure as long as it is not dangerous. Typically, the stopping of the product is supposed to be non-dangerous, if it is not a high criticality product such as an aircraft piloting system. Consequently, the implied redundancy is much smaller, and thus fail-safe techniques are often used in marketed products.

Studies on *fail-safe systems* have been promoted in the 60's by several professional institutions working for different critical applications, such as nuclear plants, terrestrial and air transportation systems. The main goal is to reduce the human losses or injuries during accidents. The raised failures must not have dangerous effects. For example, it is acceptable that a traffic

light controller be permanently blocked in a (Red, Red) state, as this failure is supposed not to be dangerous. Naturally, this failure is not minor, as it involves a great deal of disruption of the traffic. However, the drivers will pay attention. On the contrary, a (Green, Green) failure is obviously dangerous. Let us note that the technical solutions to achieve the fail-safe goal may increase the complexity of the resulting products, thus increasing the failure probability, and reducing the global reliability.

In section 17.1, we will consider the relations existing between the *risk* notion and the *safety* parameter. Then, in section 17.2 we will analyze the main techniques allowing the design and the realization of *fail-safe systems*.

17.1 RISK AND SAFETY

17.1.1 Seriousness Classes

Failures altering the behavior of a product are grouped together in *seriousness classes* according to the seriousness of their consequences:

- on the product itself (which can, for example, be destroyed or not),
- on the environment (the user).

When dealing with the consequences on the user, these seriousness classes can be based on a *quantitative evaluation*. For example, the number of casualties of road accidents: no effect (0 casualties), individual (1 casualty), at the level of a group (from 2 to 10 casualties), at the level of a state (from 11 to 1000 casualties), at the level of a population (more than 1000 casualties). The effects are thus quantified.

The definition of the classes can also be based on a *qualitative evaluation* of the effects. For example, the DO-178B standard for the civil aeronautics defines 5 classes:

- a. *catastrophic*, or *disastrous*, leading to human lives loss,
- b. *dangerous*, or *serious*, leading to a small number of casualties and/or serious injuries of passengers and members of the crew, or preventing the crew from achieving its task in a precise and complete manner,
- c. *major*, or *significant*, leading to injuries of the passengers and members of the crew and reducing the efficiency of the crew,
- d. *minor*, or *benign*, leading to upset of the passengers and a small increase of the workload of the crew,
- e. *without effects*.

These same categories can also be defined from a functional point of view, according to the degradation level of the functions of the aircraft:

- a. **catastrophic**, when the flight cannot be continued, or the landing is impossible,
- b. **dangerous**, when the reduction of the functions of the aircraft do not allow a normal achievement of the flight,
- c. **major**, when a significant reduction of the functionality of the aircraft is induced by the failure,
- d. **minor**, when the failure provokes only a partial reduction of the functions of the aircraft,
- e. **without effects**.

The placement of a failure in a class is a decision involving numerous criteria. Thus, a same failure of a product can be considered as dangerous or catastrophic according to the application using this product. For example, the stopping of the unique engine of a commercial aircraft is considered as catastrophic because this failure generally leads to the death of the passengers and the crewmembers. The same failure could be considered as major in the case of an airfighter, because the pilot can be ejected from the aircraft. Aeronautics examples may also show that the assignment of a failure to a seriousness class can depend on the operational phase during which the failure occurs: the breakdown of the engine when the aircraft has landed or is parked is a minor failure.

17.1.2 Risk and Safety Classes

At the beginning of the 20th century, the breakdown of the engine of an aircraft had the same tragic consequences as today. However, the **risk** of this failure, that is to say its occurrence probability was much greater.

For each event, and in particular for each failure, one can estimate its occurrence probability. Then, we define several classes according to several value domains of this event probability. For example:

- an event is **probable** if its occurrence probability is greater than 10^{-5} ,
- an event is **rare** if its occurrence probability is within the interval $(10^{-7}, 10^{-5})$,
- an event is **extremely rare** if its occurrence probability is within the interval $(10^{-9}, 10^{-7})$,
- an event is **extremely improbable** if its occurrence probability is smaller than 10^{-9} .

This measure grid is arbitrary. For example, the *probable* interval can be divided into two sub-intervals: *frequent* (probability $> 10^{-3}$) and *reasonably probable* (probability is within the interval $(10^{-5}, 10^{-3})$). On the contrary, other measurements group together *rare* and *extremely rare* in a class of probability belonging to 10^{-9} and 10^{-5} . Events with a probability smaller than 10^{-9} are frequently qualified as *impossible*.

The example of the first aircrafts illustrates another important aspect: the notion of *risk acceptability*. Indeed, the occurrence probability of a catastrophic event during a flight was more important than it is nowadays. This situation was however accepted by the pilots and the passengers!

We call *acceptable risk rate* the accepted maximum probability value of the failures belonging to a given seriousness class. The term *tolerable probability* is also used instead. For example, we only accept a risk lower than 10^{-7} of having a serious accident for a given transportation system.

Hence, we define *safety classes* by associating the *failure seriousness* with the *acceptable risk rate*. The resulting data define the safety requirement imposed on an industrial project: which risk level is acceptable for a given failure type. For example, the DO-178B avionics standard imposes the following values for the seriousness classes:

- the probability of *minor* failures can be *probable*, e.g. greater than 10^{-5} ,
- the probability of *major* failures must be at least *rare*, e.g. between 10^{-7} and 10^{-5} ,
- the probability of *dangerous* failures must be at least *extremely rare*, e.g. between 10^{-9} and 10^{-7} ,
- the probability of *catastrophic* failures must be *extremely improbable*, e.g. lower than 10^{-9} .

Let us note that a maximal accepted value is not a value to be necessarily reached! For example, it is positive if a *major* failure has a real occurrence probability of 10^{-8} .

It should also be noted that these standards result from a tradeoff between the safety aim and the technological possibilities offered at a given time. Consequently, these standards evolve according to the continuous evolution of the technology: thus, the limits of accepted risks in our modern societies are constantly pushed away according to the technology which makes it possible to satisfy more and more safety requirements. The considerable evolution of the safety requirements in aeronautical industry during the last century is a significant example of this positive changing. *Figure 17.1* shows the steps of the risk acceptability. In order to take the continuous aspects of the acceptability notion into account, a curve is extrapolated. This curve is called *acceptability curve*.

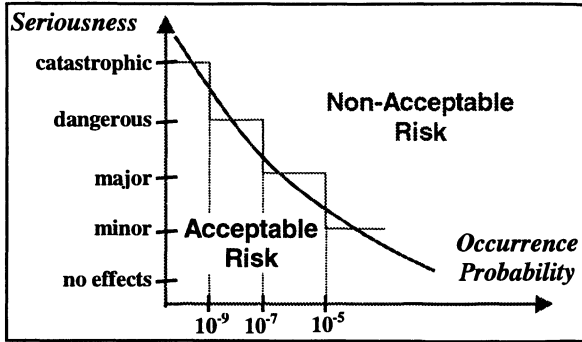


Figure 17.1. Risk acceptability

A product is qualified as *acceptable product* according to safety requirements if and only if the coordinates of all identified failures belong to the domain located above the acceptability curve. In particular, this figure shows that the risk of a failure is acceptable even if its seriousness is high, provided that its occurrence probability is sufficiently low.

This acceptability notion is often established by reference to natural risks. We hence evaluate the seriousness of an event by the number of casualties. The occurrence probability is estimated in terms of mean value of the casualty number caused by such event in a given duration. For example, here are some natural risks analyzed on a 100-year period. The resulting seriousness corresponds to the average number of dead persons and the probability is the average number of cases observed during this period:

- avalanches and landslides (400 to 4 000 deaths, 6.74 cases per century),
- floods (200 to 900 000 deaths, 37.3 cases),
- typhoons and cyclones (137 to 250 000 deaths, 37.5 cases),
- earthquakes (5 to 700 000 deaths, 330 cases),
- volcanoes eruptions (1 to 28 000 deaths, 2 500 cases).

The specialists analyze experimental data of natural risks, or risks induced by humans, and they calculate min-max intervals of probability for a given situation: for example, a person or a population faced with a disease, an industrial site faced with the falling of a meteorite, a flooding or an earthquake, a spatial mission faced with the action of radiation or heavy ions, etc. The *criticality analysis* tools introduced in the next sub-section, allow evaluating the probability-seriousness relationship associated with the failures of a product.

17.1.3 Fail-Safe Systems

We have just defined the notions of acceptability and non-acceptability domains associated with failures: for example, a catastrophic failure must have a probability lower than 10^{-9} . Now, it is important to answer the following question:

Does a given structured system resulting from a given design process and using given hardware and software technologies belong to the acceptable domain or, on the contrary, does it correspond to a non-acceptable risk?

Several **criticality analysis** methods allow to estimate the risks associated with a system, or to compare two systems implementing the same specifications. These methods belong to the two different approaches already introduced in Chapter 7: the *qualitative analysis* and the *quantitative analysis*. One of the most popular techniques in numerous industrial application fields is the **Failure Modes and Effects and Criticality Analysis (FMECA)**. This technique extends the FMEA (see section 10 of Chapter 7), taking the probability of occurrence of faults of components into account. Thus, the effects of these internal faults are propagated into the structure to deduce the product failures and their probability. Then, these results must be compared with the acceptable risks defined by the safety classes.

In numerous practical cases, the *safety requirements* cannot be satisfied in spite of the use of safe design methods. So we ask the following question:

How to realize a system or how to improve the design of a system, in order to comply with the safety requirements?

The more serious the failures are, the more important the prevention and the removal (fault detection and fault extraction) means involved during the creation and manufacturing stages of the life cycle are. However, in spite of these efforts, errors may occur during the operational stage. The reasons are twofold: the impossibility to eliminate all design/production faults, and, in the case of hardware technology, the occurrence of new faults at any time during the operation stage (according to reliability laws). We will see in Chapter 18 how to tolerate their occurrence, that is to say, to continue to offer the expected service in spite of faults. The mechanisms used to perform fault tolerance are however very complex, and their implementation is sometimes not acceptable for economical reasons, but also, paradoxically, for safety reasons! Indeed, these complex mechanisms involve new kinds of faults. When the tolerance mechanisms cannot be implemented, the client accepts the occurrence of failures, but only failures having a low seriousness. So, the main goal is to reduce as much as possible the probability of occurrence of dangerous or catastrophic failures. **Fail-safe systems** answer this need.

The methods introduced in the following section aim at mastering the fault effects in order to bring the system in an erroneous state whose consequences are as minor as possible.

17.2 FAIL-SAFE TECHNIQUES

Safety requirements can be handled by two different approaches:

- the *intrinsic safety* based on particular characteristics of the technology used,
- the *safety design* by use of *structural redundancy*.

These two aspects will be respectively examined in sub-sections 17.2.1 and 17.2.2.

17.2.1 Intrinsic Safety

Intrinsic safety of a product is obtained by constraining the development with technological solutions which are known to be safe. These solutions essentially exploit physical properties. The intrinsic safety notion is attached to numerous domestic products. A simple example is given by electric plugs which are protected from direct contact with the fingers of children. In the same way, the electrical products have different plugs, according to the electrical power characteristics: alternative or continuous current, low or high voltage, etc. Some restrictions are included in the specifications, depending on the nature of the inputs and outputs of the product, but also its behavior and its robustness to environmental aggressions. For instance, all electrical domestic equipment such as oven, microwave, toaster, fridge, etc. must comply with official security (anti-shocks) standards. Behavioral limitations are illustrated by a microwave oven that should not be able to function when its door is opened.

This intrinsic approach covers various solutions which are particular to each product. So, no general guidelines will be presented. We will only introduce three quite different examples of intrinsic safety, in order to understand this approach and to encourage its using.

Example 17.1. Railway Switches

Everyone has observed the curious shape of the manual switches used to guide the train on a railway track: they are made of a long stick supporting a heavy mass. This is an example of intrinsic safety: the weight of the mass is supposed to prevent any wrong manipulation due to a dog, a child or the

effects of the wind.

Example 17.2. Anti-explosion standards

A lot of industrial processes make use of explosive products (gas or liquids). The anti-explosion standards which are defined for physico-chemical processes guarantee that the energy used by the electric systems (e.g. a controller) is not higher than a critical threshold above which explosion risks exist, for example 100mW. The design of control systems for such processes must integrate this constraint. Consequently, we can be obliged to forbid any electrical components and to use pneumatic circuits instead. This example illustrates constraints on realization means which are defined at specification time.

Example 17.3. Safety of a Robot

The robots designed to work in interaction with humans pose important safety problems. For example, a cleaning robot must not shock people present in the room where this robot is working. A robot operating in a workshop must not hurt the human operators. These safety problems towards human beings involve the robot itself (the mechanical and electrical parts) and its control (task supervision, trajectory generation, control, etc.). Concerning the robot itself, technological solutions exist in order to guarantee the safety: no use of dangerous tools, limitation of the motion speed, and limitation of the weight. If the robot is equipped with a joined arm, the choice of the muscles and the energy used to activate them is of great importance: thus, a pneumatic flexible muscle may prove to be less dangerous than a jack or an stepping electrical motor. To caricature, a light robot made of rubber would not be dangerous at all.

These intrinsic safety means are said to be *passive safety* techniques. The passive approach does not suppress the failures, but it makes them safer.

Quite different and complementary solutions can be proposed to give safety attribute to a robot. They use *active safety techniques* at the level of the control system: use of danger detection sensors, obstacle avoidance algorithms, alarm signals provoking the stopping of the robot in case of danger. The active approach aims at mastering the effects of faults, in order to avoid their external dangerous consequences:

- use of redundant inputs to supervise the product behavior, looking at the functional environment state, and/or
- use of redundant functions to handle numerous erroneous situations.

This approach based on redundancy technique will be developed in the next sub-section.

17.2.2 Safety by Structural Redundancy

17.2.2.1 Principles

Safety by structural redundancy concerns the design stage: structural redundancy techniques are used in order to reduce the occurrence probability of failures considered as dangerous. The safety by structural redundancy techniques tries to master the dangerous failures and to neutralize them. For example, when an error occurs in a system, it rapidly evolves towards a special state such that all the primary outputs are equal to zero (typically this can mean: all output signals are switched off). Such a state is frequently judged as not dangerous to the user (outside world), as the product is inactive.

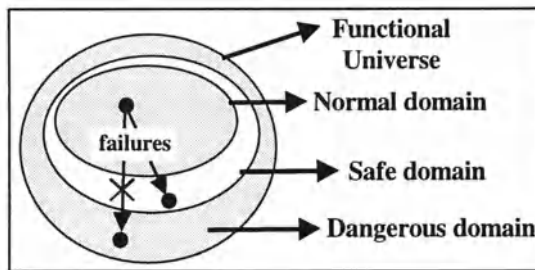


Figure 17.2. Fail-safe principles

Safety by structural redundancy is close to the fault tolerance techniques presented in Chapter 18. However, the handled problem is not expressed in the same terms. The general principles of fail-safe systems are illustrated by *Figure 17.2*. There is no obligation to maintain the delivered service. As we said before, failures can occur as long as the functioning remains in a *safe domain*. Failures leading the functioning in a *dangerous domain* must have a probability lower than a pre-defined value. Hence, we consider this approach as a step towards the fault tolerance approach.

Now we will analyze two simple but significant examples, in order to illustrate the fail-safe problems and solutions. The first example is a traffic light controller implemented with hardware technology, and the second one is a car engine controller implemented with software technology.

Example 17.4. Traffic light controller

Let us consider a traffic light controller regulating a two-way crossroad (roads A and B), as illustrated in *Figure 17.3*. The controller sends out two signals to two three-color light systems. This control is expressed here in the form of a two-variable vector (A, B) , each variable taking one of the three

values (*Green G - Yellow Y - Red R*): the output (G, R) means that the lights of the road *A* are *green* and those of the road *B* are *red*.

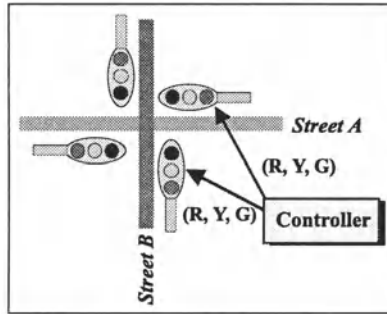


Figure 17.3. Traffic light controller

The *output static universe* of the controller has 9 vectors represented in Figure 17.4 (two independent 3-state variables). The *normal static domain* is made of 5 of these vectors corresponding to a simple evolution (we do not include the (Y, Y) state in this normal domain). The other output vectors correspond to failures. However, we suppose that the vector (G, G) is the only dangerous failure, because of an evident risk of accidents. The occurrence of this failure is not an academic case; it has been observed in the case of real electro-mechanical and electronic controllers. The other failures such that one light is green while the second one is yellow have no dangerous consequences, assuming that the car drivers are cautious when they see the yellows signals, reducing the probability of accidents and also their seriousness.

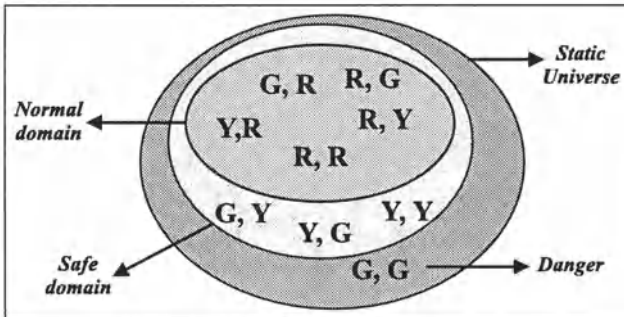


Figure 17.4. Functional domain

In order to take the safety requirements into account, it is necessary to design a controller that cannot provoke the dangerous failure, that is the state (*green, green*). We will make some assumptions to simplify this problem.

We assume that the normal evolution cycle of the controller comprises 6 states, according to the state diagram of *Figure 17.5*. Thus, the controller is a synchronous sequential system using a clock signal (*Clk*) which rhythms its evolutions. We also suppose that every output configuration different from (*G, G*) is 'safe'. The controller is supposed to be implemented as an electronic circuit, and we make the classical fault assumption of a single 'stuck-at 0/1' fault of the inputs and outputs of gates and flip-flops. Input *Clk* is supposed to be faultless.

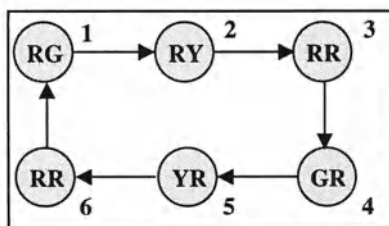


Figure 17.5. State diagram of the controller

Taking these hypotheses into account, we propose the following steps to design the controller as a fail-safe logical circuit:

- the 6 internal states are coded by means of a 2-out-of-4 coding according to the state table of *Figure 17.6*, each internal variable being materialized by a synchronous D Flip-Flop;
- each internal variable y_i , which is connected to the D input of the D_i flip-flop, is realized as an independent SIGMA-PI logical structure using no inverters (this structure is said to be 'monotonic');
- each output variable R, G, Y going to one of the two light systems is also realized as an independent monotonic structure with AND and OR gates.

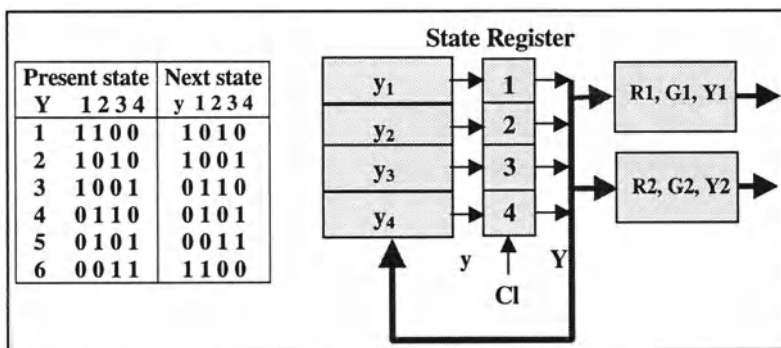


Figure 17.6. Fail-Safe design of the controller

Exercise 17.1 enters more into the details of this design method and proposes the analysis of the influence of the faults on the behavior of the circuit. We will only expose here the main results issued from this study.

Each single fault altering the sequential part (comprising the AND and OR gates and the D flip-flops) is transformed into an error that provokes an exit of the 2-out-of-4 code. If the number of bits '1' of the state register is smaller than 2, then at the next clock tick the internal state goes to the well state (0000), and the *R*, *Y*, *G* primary outputs are set to '0': hence all the lights are switched off, situation supposed not to be dangerous. On the contrary, if the number of bits '1' of the internal register is higher than 2, at the next clock tick this state will evolve towards the state (1111) which is also a well safe state. Then, the output signals are all switched on, and we suppose again that this situation is not dangerous (the car drivers will be alerted that something wrong has occurred, or we can suppose that the amplifier electronic circuit inside the light system will no drive any light in that case).

Finally, any single fault altering one independent monotonic circuit generating the primary outputs will be activated as no light 'on' or several lights 'on'.

Example 17.5. Fuel injection controller

We consider a software application which manages the engine of a car: fuel injection, spark ignition, etc. We assume that the normal control of the engine is processed after an initial phase named `Start_Engine`. If the initial phase fails, the normal treatments of the cycle must be aborted. The fuel injection must be stopped, in order to avoid a dangerous failure. Indeed, if the engine does not run while fuel is injected, then a hot engine can explode! To avoid the occurrence of this failure, erroneous events occurring during the initial phase must absolutely be detected and handled. The reaction will be the stopping of the injection function. Consequently, the product fails, but the failure is not dangerous. To guarantee the error handling, we will use an exception mechanism.

The exception mechanism offered by some programming languages has already be presented. It allows errors to be detected and signaled. When an error is raised at run-time, the erroneous component execution is stopped and resumed on an exception handler.

Here is an illustration of an exception handler associated with a procedure called `Start_Engine`:

```

procedure Start_Engine is
begin
    . . . -- body
exception
```

```

. . . -- exception handler
end Start_Engine;

```

Thus, this mechanism introduces a structural redundancy. The normal treatment (body) is separated from the treatment processed in reaction to an error (exception handler).

In terms of safety, this solution is preferable to the use of an error parameter as illustrated by the following specification:

```

procedure Start_Engine(Error: out Error_Type);

```

Indeed, this solution requires that the calling subprogram checks the value of the returned parameter. If this checking is omitted, the program execution may lead to a dangerous failure, as illustrated by the following extract:

```

...
Start_Engine(Error_Status);
Increase_Fuel_Injection;
...

```

Moreover, without exception mechanism provoking an automatic and immediate branching to a specific treatment, the disruption of the normal execution of the program must be explicitly introduced. Such a specific design is illustrated by the following program extract:

```

procedure Start_Engine (Error: out Error_Type) is
begin
    ...
    if Condition1
    then Error:= Failure_In_Electrical_Supplier;
    else ...
        ...
        if Condition2
        then Error:=Injection_Failed;
        else ...
            ...
        end if;
    end if;
end Start_Engine;

```

The resulting complexity is increased by the use of nested *if* structures to take various considered errors into account. This increases the risk of design faults and thus of failures.

Thanks to exception handlers, treatments leading the system or its functional environment to a safe state can be written. For example, the new state is assigned and the jacks of the controlled process are closed (environment). Then, the exception is propagated to provoke the same

reaction mechanism in the calling sub-program. Such a situation is illustrated by the following program:

```

procedure Start_Engine is
begin
    . . .
exception
    when Error => Shut_Down_Fuel_Injection;
                Engine_State := Stopped;
                raise Error;
end Start_Engine;

```

17.2.2.2 Fail-Silent and Fail-Fast Systems

By definition, fail-safe systems must not raise dangerous failures. One should note that what is considered as a dangerous behavior is a relative notion that depends on the application domain. In many cases, the specialists consider a 'passive state' as not dangerous: the product stays inactive or silent. This kind of fail-safe system is called *fail-silent system* or *fail-passive system*. This *safe state* suits well to control applications for which manual recovery mechanisms exist. For example, the failure of an automatic flight control can be recovered by a switch back to a manual control conducted by the human pilot. It is important that the automatic system does not continue to act on the actuators of the aircraft. The fact of forcing a failing system to enter an inactive state avoids further degradations of its environment.

The example of the traffic light controller which evolves towards a 'all 0' safe state is another example of fail-silent system. Hence, car drivers will use the crossroad as if no light control were available.

On the contrary, in other cases, a degraded minimal service is required in case of failure. For a pressurization system of the cabin of an aircraft, it is necessary to guarantee that no passenger will be injured and that the flight will pursue to its destination. Oxygen masks are therefore provided to the passengers. In the case of a failing traffic light controller, one might require that both light systems be in a blinking *yellow* state. It would hence be more expensive to achieve this goal.

The time necessary to reach a non-dangerous state such as a silent state, or to start a minimum service, is also an important attribute of fail-safe systems. When the detection of an error (erroneous state) is performed, the system can produce non-desired transient interactions before reaching the safe state. The specifications of *fail-fast systems* integrate a *maximal duration* to reach the safe state.

The use of the exception propagation mechanism (see section 2 of Chapter 14) allows answering this fail-fast need. Indeed, this mechanism can automatically propagate the error signaling from the sub-program where this

error occurred and was detected towards the main program through the calling chain. This approach is advised by the standard ISO 15942 when Ada language is used.

17.2.3 Self-Testing Systems and Fail-Safe Systems

Similar redundancy techniques are used, on the one hand, to increase the safety, and on the other hand, to detect errors on-line during the system operation (self-testing property examined in Chapter 16). However, the goals of these two approaches are quite different. *Figure 17.7* compares these two approaches, showing their similarities and their differences.

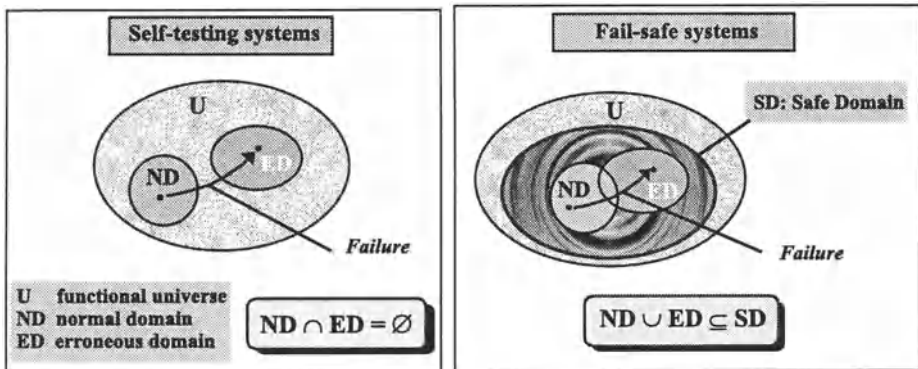


Figure 17.7. Comparison between fail-safe and self-testing systems

- 1) For a *self-testing system*, the two functioning domains, the *normal functioning domain* (*ND*) and the *erroneous functioning domain* (*ED*), must be disjoint in order to allow the detection of errors: $ND \cap ED = \emptyset$.
- 2) For a *fail-safe system*, these two domains are not necessarily disjoint, but they must be included in the non-dangerous or safe domain (*SD*): $ND \cup ED \subseteq SD$.

Hence, any self-testing system is also fail-safe if: $ND \cup ED \subseteq SD$.

We deduce from this remark that the design of fail-safe system frequently makes use of methods coming from the self-testing field. This is the case with the use of *m-out-of-n* or *double rail coding*.

Such a technique has been used to design the fail-safe circuit of the traffic light controller; we coded the internal states with a 2-out-of-4 code. The resulting circuit is then both *fail-safe* and *self-testing*. In case of error, the circuit evolves towards safe states (all 0 or all 1). Moreover, the circuit is self-testing, as the final safe states do not belong to the 2-out-of-4 codewords. The detection can lead to a later repairing.

17.2.4 Fail-Safe Applications

In this sub-section we present briefly some simplified but significant illustrations of design choices involved in up-to-date fail-safe systems.

17.2.4.1 Automotive Systems

Example 17.6. Engine control and air cooling systems

Let us first consider again a system controlling the engine of a car. It is implemented by means of a single processor executing a software constituted of several tasks managing, in particular, the ignition of the sparks, the regulation of the slow running of the engine, and some functions of the air conditioning of the cabin. The spark ignition task is activated with a period which varies according to the speed of the engine. So, the higher is the speed (number of rotations per minute), the more frequently the injection task is executed, and hence, the more it makes use of the processor. Let us note that the choice of a single processor to implement these tasks is typical of this application domain and results from strong financial constraints.

The engine can be rotating at a high speed for short durations. This is the case when the driver accelerates to perform a fast overtaking. In order to face this brutal acceleration, the designer can chose between several solutions. A first solution is to continue the execution of all the tasks, according to a circular execution: the time slice allowed to each task is fixed; the ignition task will be activated every N microsecond. Thus, the speed of the engine reaches a maximum speed which is not due to the mechanical resources but due to the software control system. On the contrary, if we decide to momentarily suspend the other tasks, the engine speed can increase rapidly because the processor is used only for this activity. This second choice corresponds to the design of a fail-safe product. Indeed, the product has a failure, as the task performing the air conditioning has been suspended during the acceleration time. However, this situation is acceptable, whereas the first solution, which limits the engine acceleration, is dangerous.

Example 17.7. Fail-silent ABS System

Antilock Braking Systems are fail-safe. Any error in the electrical system is detected, and the control unit is switched to a safe 'off' mode, allowing the conventional hydraulic brakes to be used. Such a system is *fail-silent*. To satisfy these safety requirements, a redundant duplex technique is generally used. Two ABS units run in parallel and, when the slave unit disagrees with the master unit, an interrupt lead the whole system in the safe off mode. Let us note again that the fail-safe demand does not require fault tolerance techniques to be implemented.

17.2.4.2 Avionics System

Let us consider a system embedded in an aircraft. It aims at helping the pilot during the landing phase of the flight. Let us suppose, that the software integrated in this product has a function which calculates a variable X from the knowledge of the altitude of the aircraft:

```
X := Y * Altitude;
```

This variable X is used by approach equipment.

When the value of the variable `Altitude` becomes very important, the multiplication operation may provoke an overflow. This situation is signaled to the software by an exception rising (`Constraint_Error` in Ada) if the language used offers such a mechanism. The exception handling can consist in assigning to X the highest possible value, and to resume the treatment. So, the actual behavior is equivalent to:

```
X := Float'last;
```

This design has led to a failure, as the value given to X is wrong. However, as the landing system is not being used when the aircraft is at high altitude, this failure can be acceptable: the value of X is wrong but the consequence is 'minor' or 'no effect'. Here also, we made a fail-safe design.

On the contrary, another design choice might have propagated the exception till reaching the task which is then stopped (no specific exception handler as `X := Float'last;`). This solution could unnecessarily alarm the pilot, for example by switching on a panel light indicating that the landing function has been stopped. It could also be dangerous to other tasks calling for the stopped task.

The worse solution would have been to do nothing. Then, the overflow could assign to X a random value corresponding unfortunately to a ground approach, leading to a catastrophic failure. For example, the landing flaps could be opened, or worse, the inversion of the engine may be started.

17.3 EXERCISES

Exercise 17.1. Traffic light controller

Consider the traffic light controller presented in Example 17.4.

1. Verify that the 2-out-of-4 coding allows to code the 5 internal states of the state graph.
2. Continue the coding of these states in order to obtain the logical expressions controlling the D inputs of the state synchronous D-flip-flops.

3. Verify that every single stuck-at fault leads the circuit into one of the two 'safe' internal states: 'all flip-flops at 0' or 'all flip-flops at 1'. Verify also that these two states are states from which one cannot exit.
4. What is the influence of a fault affecting the Clock?
5. Start the preceding study again with another coding of the internal states, for example a 1-out-of- n coding.

Exercise 17.2. Mathematical function processing

The development of a real-time application requires to design a task which calculates a value Y from a value X . This calculation must be realized by a fail-safe program with regard to the following error:

'the deadline of the task is reached'.

Traditionally, the implementation of mathematical functions ($Y = f(X)$) can be made according to two different approaches:

- the analytical approach which produces a result Y from the input value X , after a certain duration D ,
- the approach by successive approximations which calculates Y by means of the series $Y_n = f(Y_{n-1}, X)$.

Compare these two approaches.

Chapter 18

Fault-Tolerant Systems

With this chapter dedicated to *fault tolerance*, we reach the term of our exploration of dependability techniques. All protective fault tolerance mechanisms are defined and implemented during the creation stages of the life cycle, but their action is effective during the operational stage. They aim at guaranteeing the continuity of the service delivered by the product in spite of the presence or the occurrence of faults.

18.1 INTRODUCTION

18.1.1 Aims

A product must assure its mission in a given environment. The fault prevention and fault removal techniques allow to increase the *reliability* (reducing the probability of fault occurrence), or the *availability* in case of repairable systems (by detect-and-repair mechanisms). The *safety* criterion led us to examine the techniques related to *on-line testing* and *fail-safe design*. Now, we want to provide the designed product with the highest dependability properties by integrating mechanisms which allow the full continuation of the mission despite the presence of faults. In the best cases, these mechanisms guarantee the continuity of the delivered service without any reduction of the performances. In other cases, a possible degradation of the performances of the delivered service is accepted. The fail-safe systems constitute extreme situations. The fault tolerance techniques have a direct positive impact on the safety, reliability and availability criteria.

We insist again on the fact that one must not oppose the fault avoidance (prevention and removal) and the fault tolerance techniques. They have

complementary objectives and they are all necessary. For instance, the effort necessary to remove faults during the design process is more important if the prevention means have been neglected. In the same way, the efficiency of the fault tolerance means is based on some fault hypotheses such as the 'single fault assumption'. Hence, the use of fault avoidance techniques during the creation stages is a necessary condition.

18.1.2 From Error Detection Towards Fault Tolerance

The on-line error detection mechanisms presented in Chapter 16 are based on the use of redundancy:

- *redundancy of the function*, such as the duplex technique with the comparison of the results provided by the two duplicates in order to detect the occurrence of errors,
- *redundancy of data* representation with the typical use of error detecting codes,
- *redundancy of both function and data* when the checking of the data depends on the function, such as the case of likelihood checking.

These three situations are illustrated by *Figure 18.1*, *Figure 18.2*, and *Figure 18.3*.

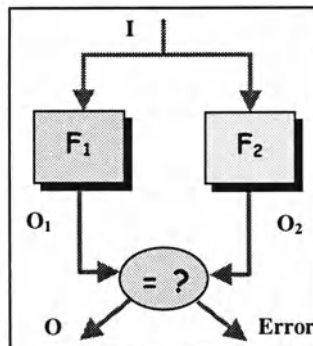


Figure 18.1. Redundancy of the function

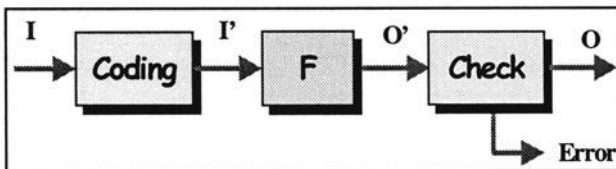


Figure 18.2. Redundancy of the data

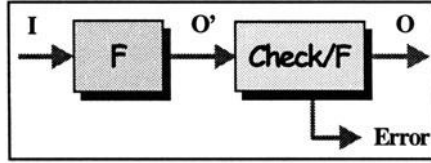


Figure 18.3. Redundancy of function & data

Redundancy is used again in order to implement fault tolerance mechanisms. Three main approaches can be considered.

First, *structural redundancy* can be used to increase the number of duplicates of the function to implement. All these duplicate modules treat in parallel the input values and produce output values; the final output value of this structure is the one which is the most frequently given by the duplicate modules. This technique called *N-Versions* is symbolized in Figure 18.4. It will be developed in section 18.2, and we will show that this technique does not require any error detection mechanism.

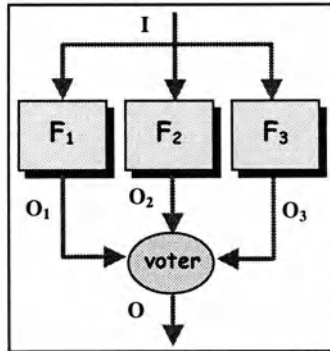


Figure 18.4. N-version

The second approach consists in executing again the same function after its first execution has reached an erroneous state. This technique called *backward recovery* is illustrated by Figure 18.5; it will be presented in section 18.3. It makes use of *temporal redundancy*, as the execution of the function is resumed from a recorded previous state.

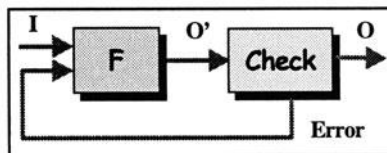


Figure 18.5. Backward recovery

The third possibility implies structural redundancy like the first approach, but here a second version is executed only if an error has been detected during the execution of the first version. This approach, called *forward recovery* (illustrated by *Figure 18.6*) will be explained in section 18.4.

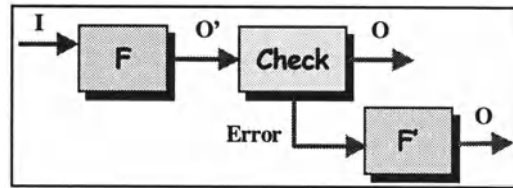


Figure 18.6. Forward recovery

So, sections 18.2, 18.3 and 18.4 respectively analyze these three groups of techniques. In particular, the implementation problems are discussed. Then, these three different approaches are compared in section 18.5. This comparison deals first with theoretical aspects: general structure of the system, nature of the redundancy involved, etc. Secondly, the three approaches are compared according to some criteria: efficiency to tolerate fault classes, extra-cost, mission duration. This analysis will allow to choose the most appropriate solution to a given application context. Frequently, the simultaneous use of several complementary techniques is necessary, in order to satisfy the set of requirements given by the specifications.

The practical use of these techniques often interferes with the design choices. For example, at which level of a hierarchical design must a tolerance mechanism be implemented: close to the technological components, or at the level of the system architecture? Moreover, the use of these techniques impacts the development process itself, making, for example, more complex the test of redundant system. These important points are considered in section 18.6. Finally, the fault-tolerant techniques are illustrated by several examples in section 18.7.

18.2 N-VERSIONS

18.2.1 Principles

The principle of the *N-Versions* technique is to simultaneously execute several samples, or *versions*, of a same functional module. If these versions have also the same implementation, they are called *duplicates* or *replicas*. The results calculated by these versions are then compared in order to produce the final result considered as correct. A typical example of this

technique is the *3-Versions*, or *TMR* (*Triple Modular Redundancy*), also called *Triplex*. This fault tolerance mechanism has been employed in some high critical applications such as spatial missions since the 70's. As shown in *Figure 18.7*, the product is constituted of three versions or modules functionally identical, and a *voter* which elaborates the final output from the three outputs of these modules. Any single or multiple functional or technological fault which produces errors altering the functioning of *only one module* is tolerated, as it has no influence on the final output.

The voter is supposed to be fault-free: its faults cannot be tolerated. For example, if this voter is implemented as an electronic circuit, any 'stuck-at' fault of its output signal may produce a failure of the product.

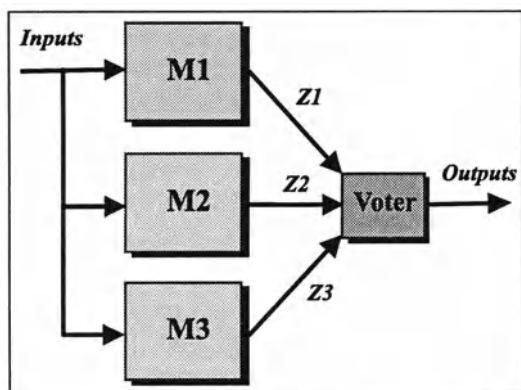


Figure 18.7. TMR or 3-version structure

So, the TMR makes use of *passive redundancy* (cf. Chapter 8) to *mask* the failure of one module thanks to the other modules (*error masking*). This approach is said to be a *compensation technique*, as the failure of one version is compensated by the contribution of the other versions.

This basic TMR structure has been generalized to the case of structured products. In that case, the TMR technique is not applied to the complete system, but to each individual module. This approach is easier and more efficient, but the local voting is more complex to manage.

18.2.2 Realization of the Duplicates and the Voter

If the fault to be tolerated is due to ageing phenomena, then it seems sufficient to use three strictly identical modules to implement the 3-versions. However, it is a wrong approach when identical stress (the environmental conditions are very similar) can lead to simultaneous faults on two or even three modules (hence, producing a violation of the basic hypothesis). More

generally, many faults, called *common mode faults*, can provoke simultaneous errors or failures of the redundant modules.

To tolerate design faults, either in the case of hardware or software technology, the different versions must absolutely be developed by independent design teams. Moreover, these independent development processes must be constrained by different design rules: for example, the algorithms and programming languages used to implement a software function must be different, or the technologies or means used to develop an integrated circuit must be different. Indeed, the used tools can have faults, or else the human developers can have the same scientific culture leading to identical faults.

Let us now consider the *voter*. This function must perform two operations: acquire the values z_i produced by the different versions, and give the final correct output value of the redundant system.

First of all, the acquisition, by the voter, of the three values z_i coming from the duplicates implies some intelligence from the voter. Indeed, the use of distinct algorithms and technologies in each version M_i implies different response times from these duplicates. Hence, the voter must manage the synchronization of the received z_i values. This voter must also detect the occurrence of a blocking of a module that will never provide any result. This situation arises, for example, when the program implementing a version M_i executes an infinite loop due to a fault.

When the values given by the outputs z_i belong to a finite domain, then the *determination of the final output value* z generally implies a majority vote: a value produced at least twice is considered as correct. On the contrary, if the z_i values belong to real numbers, the algorithm used to calculate the final output is more complex. Indeed, different but close values, such as '14.31', '14.30', and '14.32', can be considered as correct, even if they are not strictly identical. The results provided by the different versions can be analyzed: calculation of their distance to the average value, in order to eliminate possible incoherent values.

This complexity of the functions realized by the voter can also be encountered in the case of discrete values. For example, if the output values are 'colors' coded by integers from 0 to 65 535, one can consider that the values '271', '273' et '274', respectively provided by 3 versions, define the same color because of different algorithms used to calculate them.

Thus, the election, by the voter, of the correct result requires an analysis which is proper to each application: it is a 'context dependent solution'. We have noted that it is not always possible to simply elect the value appearing twice or more, as this situation may never occur. The easy solution consisting in providing an average value of the given z_i , is generally not acceptable; it must be proven that this value is significant. For the preceding

example, one of the received values, and not the average value, could be retained as a final result. Indeed, $z_1 + z_2 + z_3 = 271+273+274 = 818$ cannot be divided by 3. Hence, 273, which is closer to the median value, may be adequate.

18.2.3 Performance Analysis

18.2.3.1 Quantitative Analysis

The TMR is taken again as a reference example of the N-Versions technique. We assume that all the versions (modules $M1$, $M2$ and $M3$) are identical and have no design faults. The TMR is then used to cope with ageing faults. Their reliability is only influenced by technological problems and supposed to follow simple exponential law. The voter is supposed to have a failure probability much lower than the failure probability of each version. Moreover, we assume that the technological implementations of the three modules are different, in order to reduce the probability of having multiple similar faults. We should mention that these remarks have already been formulated when dealing with the self-testing *duplex* structure. Now, if we add the hypothesis that the three modules have exponential reliability laws $R(t)$ with identical failure rates, λ , then the resulting reliability law of the global system is given by the expression:

$$R_{TMR}(t) = 3 R(t)^2 - 2 R(t)^3 = 3e^{-2\lambda t} - 2e^{-3\lambda t}.$$

This result is obtained thanks to the *reliability block diagram* method examined in Chapter 7 and applied to redundant structures. Annex B analyzes the reliability of some significant redundant structures.

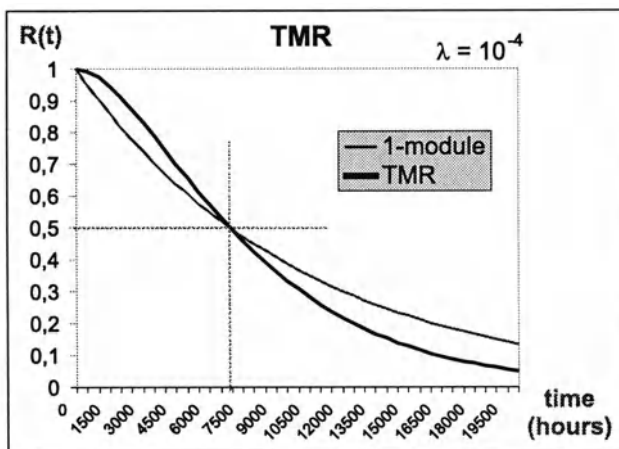


Figure 18.8. Reliability efficiency of the TMR

The reliability curve is drawn in *Figure 18.8*, for $\lambda = 10^{-4}$. We observe that for $t = 0$, the tangent is horizontal. Hence, the reliability of a TMR is better than the reliability of one module as far as the duration of the mission is smaller than approximately $t < 7\ 500\text{H}$, corresponding to the intersection between the two curves, that is $R(t) \geq 0.5$. For missions having a duration greater than this critical value, the one-module has a better reliability than the TMR. The calculations show that the MTTR of the TMR is only 5/6 of the MTTR of one module.

18.2.3.2 Qualitative Analysis

The main drawback of this technique is the impossibility to know the internal state of the system from the outside. This is said to be a ‘bunker’ structure. Indeed, an external observer cannot decide if one of the modules fails or not. As one of the negative consequences, the production or the maintenance testing of a TMR structure is not easy.

The TMR uses a passive masking technique; this does not reduce the occurrence of faults. As the global complexity is approximately multiplied by 3, the number of faults is also multiplied by 3. For long duration mission, it has been shown that passive approaches are not efficient, because of the risk of fault accumulation along the mission, and consequently the risk of provoking the violation of the basic hypothesis: only one module has a failure at a given time t . Thus, much more efficient active techniques have been developed instead.

18.3 BACKWARD RECOVERY

18.3.1 Principles and Use

The principle of the *backward recovery* technique is to resume the execution of a module M after an error has been detected by the checking of its output variables. This technique is represented by *Figure 18.9*. Thus, in case of error, we return to the past of the system, to start again its execution from a safe previous state. A trivial example of such a recovery is the *reset* of the module before resuming its functioning.

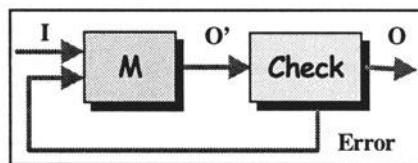


Figure 18.9. Backward recovery

The main difficulty of the backward recovery implementation comes from the choice and the management of these safe states called *recovery points* (also called *retry points* or *rollback points*). These states must be saved when they are reached, to be restored later, when an error occurs. We analyze the saving and restoring mechanisms in sub-section 18.3.3.

One typical application of backward recovery technique deals with the tolerance of transient faults in electronic circuits. The transient fault occurring at the first execution is supposed to have disappeared at the second execution. For example, a simple reset, when possible, can be sufficient to resume the proper function of such a product subjected to transient faults.

This technique has been adapted to the software technology as *retry mode*. It is mainly used to tolerate transient faults on parameter values: the procedure P is executed a second time after the acquisition of new values of the parameters, given by actuators or provided by the user.

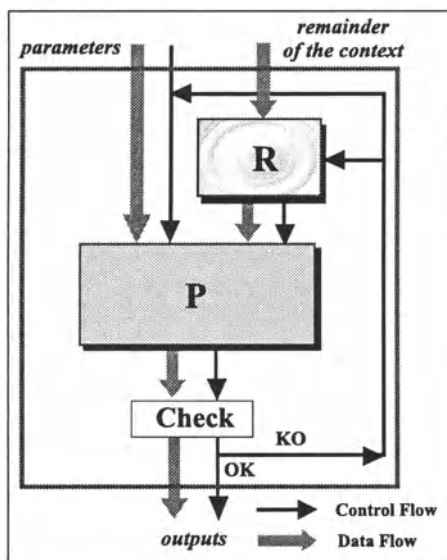


Figure 18.10. Retry mode

The implementation of this technique is not as easier as it seems to be at first. Indeed, the first execution of P can have affected some elements of the context, others than the parameters, as for example the global variables. In that case, it is absolutely necessary to also save these variables before any re-execution of the procedure P . Thus, we must save all altered variables with the parameters for their ulterior restoring. Let us consider as an example a procedure P calling several sub-programs belonging to a package (or a library). The execution of these sub-programs during a first execution of P can have modified the local variables of this package (or this library). We

are then obliged to restore their initial values before a second execution of *P*. The means described in the next sub-section are dedicated to the management (save and restore operations) of this state of the 'context'.

Figure 18.10 illustrates the principles of this tolerance technique. Procedure *P* is the module that must be made fault-tolerant, *Check* is the error detection function, also called *acceptance test*, and *R* is the component in charge of the recovery operations (save and restore). Exercise 18.6 proposes the study of an example of fault-tolerant acquisition program using this technique.

18.3.2 Recovery Cache

Backward recovery requires the implementation of saving and restoring mechanisms of the execution context. One of the most popular techniques is named *recovery cache*. After an error has been detected, this mechanism allows the previous stored context to be restored. A context is defined as the system *execution state* at a given time. In practice, it is implemented:

- by a set of values of the variables, the instruction pointer and the stack pointers for software products,
- by the states of the physical devices for hardware products.

Several recovery cache strategies exist to store and give back the state of the recovery points. Two of them will be illustrated hereafter on software technology even if their principles are more general.

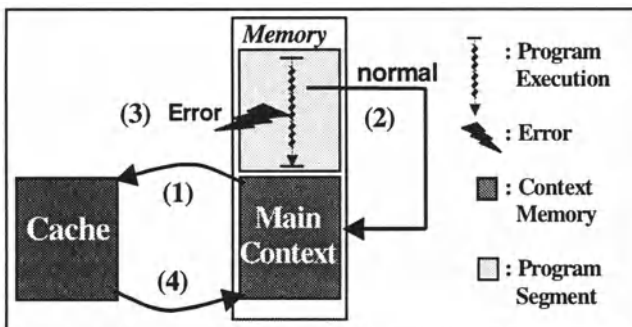


Figure 18.11. First backward recovery implementation

In all cases, the context data are copied from the main memory (i.e. the variables used without fault tolerance mechanism) in a cache which is a specially allocated memory part. This copy is labeled by (1) in *Figure 18.11* and *Figure 18.12*. We will analyze in section 18.3.3 at which instants this saving operation is performed, that is, the choice of the recovery points.

First strategy

The assignment of the variables is done in the main memory during normal processing, as shown by arc (2) in *Figure 18.11*. When an error is detected (3), the values stored in the cache are transferred to the main memory (4) to restore the previous context.

Second strategy

Another possibility consists in adopting a symmetrical strategy. During execution, the new values of the assigned variables are temporarily stored in the cache (2). If an error is detected (3), the main memory already contains the initial values. So no actions are necessary to restore the initial context. On the contrary, if the execution is successful (4), the memory must be updated (5).

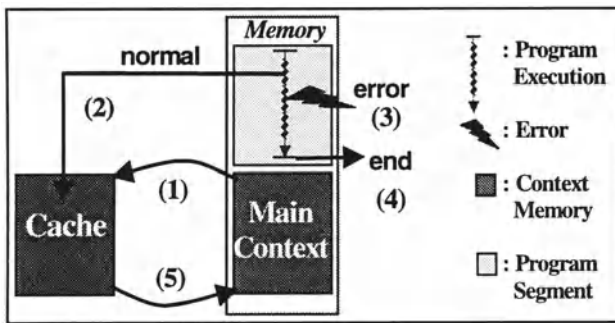


Figure 18.12. Second backward recovery implementation

The advantage of the second technique is that it only modifies the values in the main memory if there are no errors. Thus, there is no degradation (even temporary) of the integrity of the data during execution. However, the first technique is quicker if no erroneous states are reached, because it does not require updating at end of execution.

18.3.3 Recovery Points

18.3.3.1 Context Saving

It is generally not easy to choose the recovery points at which the contexts will be saved.

- This choice can be independent, both from the function executed by the system, and from the structure of this system. This is the case, for example, when a periodic saving of the current state of the system is performed. For a program, we make what is called a *snapshot* of the

memory. This image will then be restored if an error occurs before the end of the pre-defined period.

- The recovery points can also characterize the important states of the system. Hence, the recovery of these states only requires the storage of some characteristic data.

If the first solution is easier to implement, it is certainly the less efficient. Thus, during a long duration with a small activity of the system, numerous save operations will be made whereas most of the values have probably not been changed. On the contrary, the system can evolve through numerous states during a period separating two save operations. This situation occurs during traffic peaks implied by fast evolution of the environment of the system. In such a case, the restoring of the saved context will bring back the system in a state far from the state reached when an error has been detected. In many applications, critical phases are specified, for which the recovery must lead the system in a state close to the one reached before an error appeared. A periodic save operation cannot satisfy this requirement if the period is too long. If the period is too short, this technique is untractable, because the system overhead to perform the save operations is too important.

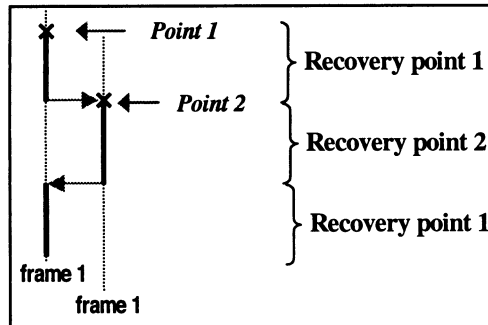


Figure 18.13. Frame nesting and recovery points

Moreover, the management of the memory used for context saving at recovery points is generally not limited to the assignment of a static memory area and the substitution of the old values by the new ones. More complex mechanisms must frequently be implemented. This is the case for software using nested frames (subprograms, blocks, etc.). Thus, the hierarchical structure of the programs influences the recovery technique implementation. If the recovery point denoted *Point 1* exists in a first frame (see Figure 18.13), and if this frame calls a second frame which defines its own recovery point *Point 2*, then, when exiting the second frame, the recovery context of the calling frame must be restored (*Point 1*). This obviously requires a stack mechanism to manage the recovery caches.

18.3.3.2 Context Restoration

Sometimes, the mechanism used to *restore the context* is much more complex than a simple copy back of the values saved at the last recovery point. Such a situation arises, for instance, in real-time applications implemented by several tasks or processes. Let us consider the example shown in *Figure 18.14*. It represents from left to right the evolution of the execution of three processes: *P1*, *P2* and *P3*. A character 'X' signals the moment when the corresponding process context is saved. The vertical dotted lines specify the communication or synchronization times between two processes.

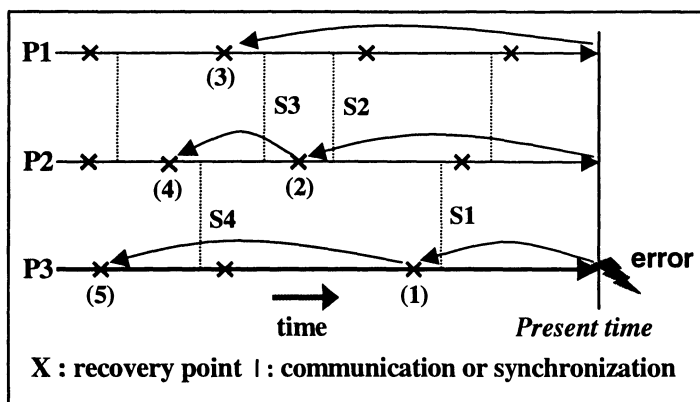


Figure 18.14. Domino effect

If an error occurs at 'present time' during *P3* process execution, then the context of this process execution must be recovered in (1). As a communication or synchronization *S1* occurred during the time intervals [(1), present time] between process *P3* and process *P2*, the resumption of *P3* requires to go backward through process *P2* to its last recovery point (2) before time (1). Process *P2* handled a communication or a synchronization with *P1* (*S2*), so the backward movement in *P2* must generate a backward movement in *P1* to the recovery point (3). This implies backward movement of *P2* to (4) (due to *S3*) and then backward movement of *P3* to (5) due to *S4*.

This phenomenon is called *domino effect*. Mechanisms have been proposed to handle this phenomenon. We will not present them here, as this book does not cover distributed systems.

18.4 FORWARD RECOVERY

18.4.1 Principles

The *forward recovery* technique consists in resuming the execution of the system in a new state after the detection of an error. This state is qualified as 'new' if it has not already been reached during the past execution. The *state* of the system is characterized by the values of its inputs and outputs, and also its internal functioning state. Any never assigned value of one of these elements (inputs, outputs and internal state) leads to a new state. Two situations are often considered:

- return to previous input values, but with new internal state of the system functioning (*recovery blocks*),
- preservation of the current value of the outputs and evolution of the system functioning into a new internal state (*termination mode*).

We will now study these two situations in the following two sub-sections.

18.4.2 Recovery Blocks

The *recovery blocks* technique makes use of a passive redundant module (or component) which is activated when an error is detected in the first module. This approach is illustrated in *Figure 18.15*. The thick gray arrows which represent the data flow show that the redundant module Q starts its execution with the same data as P (the execution of which has been detected as erroneous).

This technique requires a backward recovery of the input data, implemented, for example, by a recovery cache. However, globally, the recovery may be considered as *forward* because a new module is used; hence, the internal functioning state is new. If the implementation of the modules is made by a software technology, this new internal state will contain the address of the redundant subprogram (Q) that offers the same functionality as (P).

We have just considered the case of a passive redundant module. In order to tolerate the occurrence of an error in this second module, it is necessary to use a third module, and so on. These spare modules are also called *alternates*. The word *version* is also used, but this word does not explicitly express that the executions are alternative (P then Q) and not simultaneous (P and Q).

The technique of *recovery blocks* is generally considered as a backward recovery technique. This is true from a functional viewpoint of the system, or if a hardware implementation based on two identical components is made.

Indeed, in these two cases, P and Q are the same. On the contrary, software implementation of P and Q are generally different. This discussion highlights the limits of our general presentation. So, recovery blocks can be introduced conventionally as a backward recovery technique, signaling the particular case of software implementation.

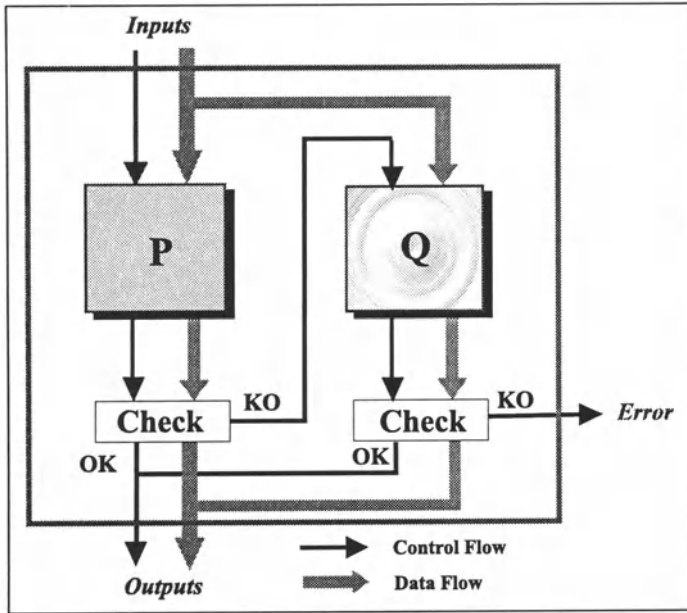


Figure 18.15. Recovery blocks

18.4.3 Termination Mode

The *termination mode* technique consists in finishing the processing started by a first module (or component) P in which an error is detected. Figure 18.16 illustrates this technique. We see that Q achieves (or completes) the activity actually realized by P , without starting with new initial data but by using the current context. The data remains the same, as Q uses the outputs issued from P , but the internal state of the functioning is new, as Q replaces P .

The termination mode is, for instance, implemented by the exception mechanism of the Ada language. P is the current treatment and Q is the exception handler:

```
begin
  P;
exception
  when others => Q;
```

end;

When an error is detected in the execution of the *P* statement block, an exception is raised in order to signal this error. The processing is resumed by execution of *Q*. The values of the variables modified by *P* are preserved (no backward operation), while the instruction pointer has a new content (the memory address where *Q* starts).

A second example of fault-tolerant mechanism based on the termination mode is given by the on-line use of error detecting and correcting codes. Indeed, the results produced by a module *P* are detected as erroneous by a code violation, but they are exploited by a corrective function *Q* (program or circuit) in order to produce a correct final result.

In these two different examples (Ada exceptions and EDCC), the treatment does not return into any previous state. In particular, the initial input values are not re-used to handle the error. Even if the chosen state used to resume the execution preserves the data obtained by the erroneous module (outputs of *P*), the internal execution state is new, branching to (*Q*). Thus, this technique is a forward recovery technique.

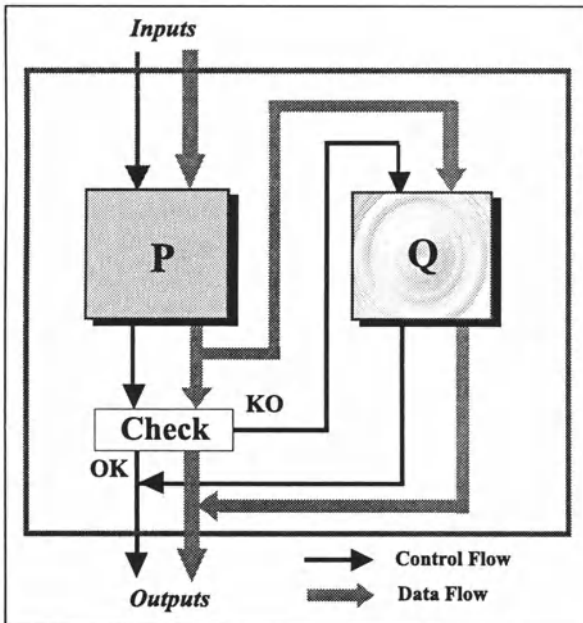


Figure 18.16. Termination mode

18.5 COMPARISON

The developers can choose between numerous and varied tolerance techniques to design a specified high-dependable product. We must compare their similarities and their differences to help for a better choice. In practice, it will frequently be necessary to combine these techniques to deal with many different classes of faults that have to be tolerated.

18.5.1 Similarities

18.5.1.1 Redundancy

All proposed fault tolerance techniques have an essential common feature: they are based on the use of *redundancy*. The dependability of a system evolves from 'fault detection' towards 'fault tolerance' by a global increase of its redundancy. In all cases, this added redundancy is however *structural*.

The redundancy involved by the *N-versions* approach is *passive*, as the $(N-1)$ replicates modules can be suppressed if no faults alter the first module.

The *forward recovery* approach is also based on passive redundancy, as the new module (Q) to be executed in case of error in the first module (P) can also be suppressed as far as no error occurs.

Another attribute characterizes the fault tolerance mechanisms: the presence or the absence of error detection means. The tolerance mechanisms that make use of an error detection action followed by a recovery of the error are qualified as *active tolerance*. On the contrary, if no error detection is explicitly needed, the *tolerance* is said to be *passive*. All techniques presented here, except the N-Versions, belong to active fault tolerance.

18.5.1.2 Tolerance Mechanism Framework

Even if each technique presented in the preceding sections seems specific, it belongs to a very general framework, which involves three synchronized stages:

1. *error detection*, by on-line testing techniques previously studied,
2. *error diagnosis* by *localization* of the failing module (determination of the *state of the degradations* caused by the fault just before the error detection),
3. *error recovery* by the following sequence:
 - *protection* against further propagation of the detected error by *anti-contamination* process,

- then *error correction* and/or evolution towards a *safe state*,
- and finally *reconfiguration*, leading to resume the normal activity of the product.

In the case of the N-Versions technique, the two first steps do not exist since the error recovery is performed by *compensation*. The successive degradation of the modules during a mission can lead to a 2-version system which is no longer fault-tolerant. The introduction of error detection into the N-Versions structure, proposed hereafter (such as the NMR technique) will provide a useful knowledge of such degraded situation.

In the two other techniques, the detection and the diagnosis are completed by a backward or forward recovery. It seems curious to use error diagnosis in the case of backward recovery since the corrective treatment starts from a previous state. However, the effects of the erroneous treatment must be corrected before returning to any previous state. For example, if a resource has been accessed but not released, this release must be done before the recovery process calls again for the resource; thus, we avoid a deadlock situation.

Depending on the alternative treatment Q , the tolerance mechanism can either:

- restore the entire ability of the product to deliver its service,
- or proceed to a *progressive and graceful degradation* of this service when the available resources are not longer sufficient.

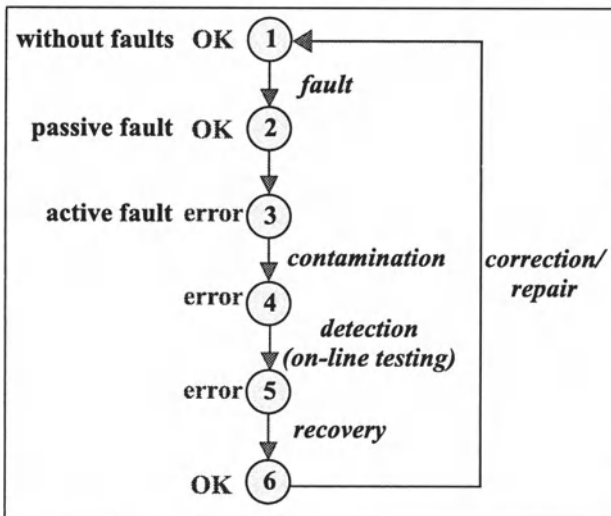


Figure 18.17. Reconfiguration process

Figure 18.17 illustrates such general active reconfiguration process. Initially in a faultless state (state 1 of the graph), the product is affected by a passive fault (state 2), and then this fault is activated as an error (state 3). This error is propagated by contamination in the product structure (state 4), until an integrated on-line testing mechanism detects one of the active errors (state 5). Then, the contaminated part of the product is isolated to block any further error propagation, and a rescue service is provided (state 6) until a correction/repair operation allows to restore the initial capability of the product (state 1).

18.5.2 Differences

The choice of a tolerance technique must take several criteria into account: the fault classes to be tolerated, the duration of the mission, the 'know how' and the mastering level of the development team, and the impact of the technique on the development process.

18.5.2.1 Faults to be Tolerated

Each one of these techniques suits to the tolerance of some particular classes of faults. It is thus important to know which fault classes are most probable in a given project.

Techniques based on the backward recovery are well adapted to transient faults which have then disappeared when the module is executed again. Such approach is then not relevant to handle design faults. However, a particular case must be mentioned: applications very sensitive to values of parameters, which have a very low probability. For example, a real-time application can reach an error for some very particular configuration of its tasks. Such a situation can have a very low occurrence probability. For example, the error is reached when a temporal parameter has a precise value. For instance, to avoid such error, the time between the arrival into a internal state and the date of occurrence of an external event must not be between 0.07 ms and 0.08 ms, knowing that this event occurrence date is always between 0.00 ms and 10.0 ms. If these two dates are not correlated, the occurrence probability of this external event when the system has reached the specified state is 1/1000. If this state is reached scarcely in a short duration mission, the occurrence probability of the error is low. In this case, a simple retry in a previous state has little chance to conduct again in the erroneous state.

Forward recovery techniques are well adapted to tolerate design faults, as another module replaces or completes the erroneous treatment. Naturally, the redundant module must have been developed by an independent design team with different realization constraints that prevent the introduction of identical faults in the various modules. If the recovery blocks are implemented with

two identical systems ($P=Q$), then no design faults are tolerated (cf. the analysis of the failure of Ariane 501 in Appendix D). The only faults to be effectively tolerated in that case should be those due to the ageing of the electronic components. However, the stress puts on the product can bring simultaneous faults of the hardware components used by the alternates P and Q , making thus again inefficient the fault tolerance mechanism.

In the same way, the N-Versions technique does not tolerate design faults of identical versions, or *common mode faults*, such as simultaneous faults due to operational stress (temperature, etc.). The TMR makes the fundamental assumption that only one module can fail at the same time (single fault assumption).

18.5.2.2 Acceptable Extra-Cost

During the development stages

The development of a fault-tolerant system is very expensive: choice of one or several specific techniques, design and verification of the system.

We have shown that fault-tolerant systems involve redundant components. This implies quantitative extra-costs due to supplementary components. Qualitative difficulties are added to these quantitative aspects: for example, the realization of a fault-tolerant program based on the termination mode is complex, as the primary program P detected as erroneous can be stopped at many different locations during its execution. Hence, the program Q that must achieve the task started by P must behave differently according to locations as well as the nature of the detected error.

The example of the termination mode also shows that the implementation of a fault tolerance technique influences the design choices. For example, if a program P delivers a result, an iterative algorithm is more suitable than an analytical resolution to implement a termination mode. Indeed, if an error occurs in the first treatment (P), the approximated result can then be taken as input of the second treatment (Q) to produce the precise final value. On the contrary, analytical algorithms generally produce no intermediate results that can be properly exploited by a second algorithm.

The development of fault-tolerant applications requires also designing the elements which are associated with the chosen technique: the voter for the N-Versions, the recovery cache mechanism, the recovery point management for the backward recovery, etc.

Finally, extra-costs and difficulties are added to verify the redundant system. Two problems are encountered: the testing of the elements (versions, etc.) of the application, and the testing of the fault tolerance mechanism. The first difficulty is illustrated by the test (design, as well as production or maintenance test) of TMR structures whose failures of the modules are masked. Consequently, a fault may exist in one of the three versions, which

cannot be detected at the outputs of the system, because of the fault-tolerant mechanism. In such a situation, the product is accepted, whereas it will not tolerate the occurrence of a second fault on another version. In fact, the compensation mechanism does not use any error detection techniques.

The second difficulty concerns the fault tolerance mechanism testing. This difficulty can, for instance, be due to the necessity to activate faults in order to test the error detection means. Of course, the product is designed to prevent faults, so more complex methods and devices have to be used, such as the fault injection techniques which artificially add faults.

During the fabrication and the operation of the product

Extra-costs are also implied by the fabrication of a fault-tolerant product. For example, each sample of an electronic product implementing a TMR technique will cost at least three times the cost of a basic component. In the same way, every fault-tolerant software requires more memory space to be embedded and executed.

The overhead due to the fault tolerance mechanisms can be expressed in terms of supplementary execution time (CPU time), or necessary increase of CPU performances to treat these mechanisms. We have noticed the complexity, hence the duration, of the implementation of the context save and restore procedures involved by the backward recovery techniques. When time constraints belong to the specifications, and when faster components (such as CPU) cannot be used for technological or economical reasons, the redundant modules must realize approximated calculus. The recovered function is hence degraded. The N-versions seems to be the more efficient technique since all versions are active. N processors are then necessary for efficient concurrent executions. This efficiency is to be paid by an extra-cost in terms of electronic components (redundant processors).

The economical aspects of the fault tolerance are not to be neglected. There are domains where quite expensive redundant solutions are accepted (military, avionics, spatial). Will they be always accepted? In other cases, where cost reduction is essential, only simple solutions can be envisaged. For example, the re-initialization of a system and the re-starting of the whole processing is a crude but cheap implementation of the retry mode technique. Obviously, such a technique treats the transient fault classes (due to parasitic interference, or a rare blocking situation). Methods and technology choices always result from trade-off between economical constraints and dependability performance.

18.5.2.3 Mastering the Development and Mission Duration

The designer must know that the complexity of the fault tolerance means can introduce new faults. This complexity is due to the fault tolerance

mechanism itself, but also to its impact on the design process. This last point will be considered in section 18.6.

In fact, the faults handled by the tolerance mechanism and the faults due to the complexity of the implementation are coupled. For example, the use of the N-versions technique by inexperienced development teams is not advised: the increase of the global complexity of the system (versions, voter and integration of the modules and the voter) can bring new faults added to those of the versions. On the contrary, the use of identical electronic boards in order to tolerate transient and localized hardware faults (on one board only) is efficient if it results from a well-mastered manufacturing process.

The duration of the mission must also be taken into account to choose a technique. For long duration missions, it has been shown that the TMR approach is inefficient because the progressive accumulation of faults can invalidate the fundamental single failing module assumption.

By the way, let us again stress on the fact that fault tolerance mechanisms do not prevent fault occurrence during the operational stage. On the contrary, in the case of the TMR, the probability of faults is about three times greater (the complexity is three times greater). Also, the occurrence of errors at the output of the modules is not reduced; *masking* technique only prevents final output errors, i.e., failure occurrence.

The development of redundant components simpler than the original ones is frequently justified by this need to minimize the fault risk as well as by the need to reduce the execution times. For instance, the alternate module in a recovery block technique will offer a degraded but safe service. This approach is well known by tennis players who take less risk during their second service ball.

18.5.3 Use of Multiple Techniques

Practically speaking, different techniques are often combined together to implement a fault tolerance strategy. For example, if transient faults are the most probable faults altering an electronic component, a backward recovery technique is interesting. It can be completed by an approach based on forward recovery to treat the case when several successive trials fail. We will mention some other examples.

18.5.3.1 An Extension of the TMR: the NMR

The voter implemented in the N-versions approach has to determine the correct result from the outputs given by the versions. Frequently, this function could also be exploited to reveal the incorrect results given by the versions, thus detecting their bad functioning. In the case of a majority vote, all results different from the majority of the produced results are considered

as false. When the values produced by the versions belong to a 'dense' domain, the conclusion is not so obvious. We have already mentioned that the 3 values 14.0, 14.1 and 14.2, although different, can be considered as correct if the acceptable calculation uncertainty is 0.2.

Let us now consider a TMR. We add to this structure a detection device which compares the outputs z_1 , z_2 et z_3 , two by two, and signals an error as soon as a difference is detected (see *Figure 18.18*).

The logical function performed by this detection block is:

$$error = (z_1 \oplus z_2) + (z_1 \oplus z_3) + (z_2 \oplus z_3) \quad (\oplus \text{ is the XOR operator})$$

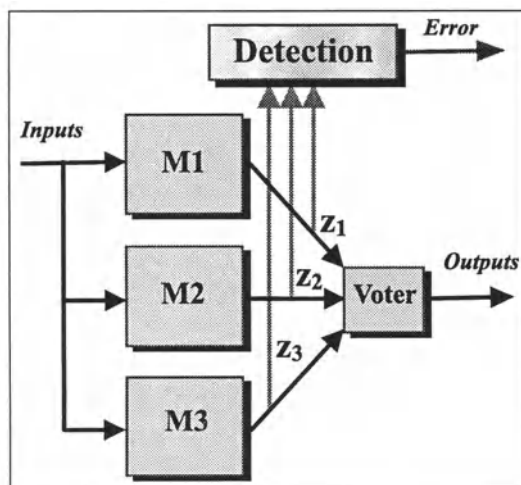


Figure 18.18. Improvement of the TMR: error detection

Still continuing to deliver a correct final output, this system is able to signal any error of one module. We have thus given to the TMR passive redundant structure an on-line testing property. Moreover, it is now easy to localize the erroneous module (this will be studied in Exercise 18.3).

We finally reach the *NMR* (*N-Modular Redundancy*) by adding several spare modules, and a *reconfiguration* mechanism which neutralizes the failing module and replaces it by a spare module. Hence, we have combined the TMR with a recovery block. *Figure 18.19* shows the principle of this structure: three modules M_1 , M_2 , M_3 , are active (they participate to the vote), while $(n-3)$ modules are waiting. When the detection circuit observes an error, the failing module is identified, and a recovery procedure excludes it, for instance by switching its power off, and activates one of the spare modules that now participates to the vote. Reliability and availability calculus show that the NMR is better than the basic TMR.

Numerous variations and extensions have been imagined from this basic scheme. In particular, some of these modifications concern the functioning mode of the spare modules. When dealing with electronic components, the best solution would be to keep the spare modules inactivated or *cold standby*; these spare modules should be stored outside the product, in good environmental conditions (temperature, protection against aggressions) in order to preserve a better reliability than the one of the active modules. This idea is not always excellent because when a spare module has to replace a failing one, it is necessary to insert it and to initialize it by restoring the reached context (cf. recovery points and recovery cache sections). For systems implementing software, this operation is relatively long, the resulting delivered service being slowed down.

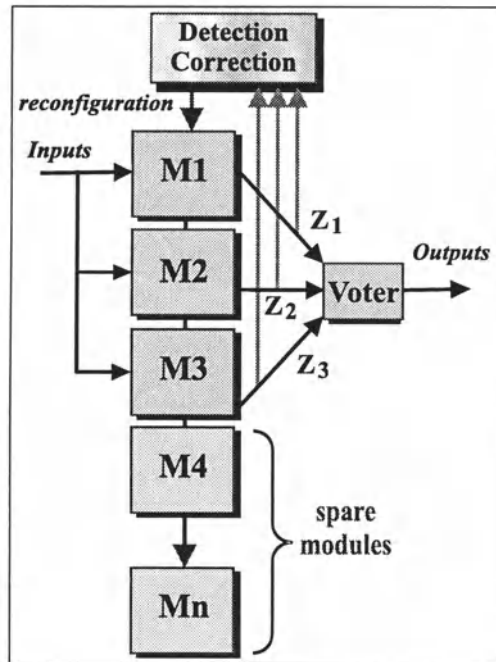


Figure 18.19. NMR

In many present systems, the cold standby technique has been improved by the use of *hot swap* modules. The faulty module can be replaced in-line, without switching off the power of the system. The new inserted module is rapidly and automatically initialized.

In more critical applications, one often prefers solutions using *hot standby* modules. In this case, the redundant modules get the current evolution of the context. The electronic components are powered on and

they receive and treat the input signals; however, they do not participate in the final vote. When an error recovery occurs, any spare module is in the same state (flip-flops, registers, variables, etc.) as an active module; hence, it can easily be switched as an active participant of the vote (by a simple electrical switch) without any initialization procedure.

This approach is used by a lot of recent modular, flexible and efficient techniques, offering fault-tolerant systems using *active redundancy* with *reconfiguration*. The possibility to integrate continuous or discontinuous on-line testing mechanisms inside the modules lead to numerous redundant architectures based on vote and switch operations: *double-duplex*, *2-out-of-4*, *3-out-of-4*, etc.

18.5.3.2 Structure with Adaptive Vote

The *structure with adaptive vote* is a particular case of the N-Version technique. When an error occurs at the output of one module, this module is switched off, the vote function being now performed on the $(n-1)$ remaining modules. This degradation process is continued as far as there are still at least three functioning modules.

This idea has been adapted to various techniques such as the *self-purging*, the *Shift-Out*, etc. The *self-purging* technique is explored by Exercise 18.5.

One can also implement such an idea in the case of distributed tasks on a computer network. Other combinations of techniques will be presented in section 18.7.

18.6 IMPACT ON THE DESIGN

The high complexity of the design of the mechanisms necessary to implementing fault tolerance techniques has already been signaled several times. Unfortunately, difficulties encountered during the development of a fault-tolerant system are not restricted to specific problems relevant to these techniques. Other difficulties result from architectural choices. Without entering the details of such problems, we would like to illustrate one of these aspects: the choice between *confinement* and *error propagation*.

Let us consider a system's component introduced during a design step. If an error is detected during the execution of this component, two actions can be envisaged:

- the error is locally handled: the error is confined into the component,
- the error is communicated to the 'parent component', that is to say the component who used the erroneous component: the error is then propagated.

The aim of the *confinement* is to avoid:

- collateral effects on the other components introduced at the same design level that the erroneous component,
- parent effects on the components belonging to higher hierarchical levels (parent, grand-parent, etc.),
- effects on the resources used by the components, such as the hardware (microprocessor and memory) and the software elements (operating system).

By avoiding this *error contamination*, we try to avoid the failure of the global system.

Confinement of the effects of faults/errors on the resources is a more and more important need, as recent systems use a same plate-form to execute several various functions of one application. Formerly, a natural confinement resulted from the fact that each function was executed on an independent plate-form. This new situation is for example encountered in embedded avionics systems: the *Integrated Modular Avionics* concept.

Confinement means must prevent:

- *direct contamination*,
- *indirect contamination*.

As example of direct contamination, an erroneous task *T2* assigns a wrong value to a variable shared with another task *T1*. Indirect contamination is illustrated by a task *T2* that enters an endless loop because of an error and does not release the processor, causing a schedule failure of another task *T1* (starvation phenomenon).

Different and independent of the confinement mechanism, the *error propagation* mechanism aims at signaling an error occurrence. This propagation is justified if the erroneous state has been reached, not because of the component itself, but because of its external environment, or because of the other components of the system. For example, a subprogram can behave incorrectly because of the way it is used: incorrect values of the calling parameters, inadequacy of the state at the calling time, etc. For example, the call to a *Pop* function applied to an empty stack cannot be correctly treated by this function. However, the caller is not necessarily informed of this situation if this empty state is an internal attribute. The *Pop* subprogram detects an error of which it is not responsible. Consequently, this program must propagate this error.

On the contrary, if a call to a *Push* subprogram is made when the stack is full, two design choices can be made:

- *error confinement* by increasing the stack size in order to be able to store

the value passed to the *Push* program as a parameter; we then consider that the fault is due to a bad estimation of the stack size necessary to the execution;

- *error propagation* by signaling the impossibility to execute the function; the fault is then attributed to the component using the stack.

Frequently, these two possibilities are simultaneously used: the component detecting an error makes a part of the error recovery, while communicating the error to the user component for complementary actions. For example, if the component is a subprogram, its partial execution before the error detection can have modified some local or global variables; hence, these variables must be restored to their initial values before the signaling of the error to the calling program which will make other actions.

Whatever the choice, a decision must be taken by the designer. The worse case would be:

- not to explicitly signal an error which is not fully treated in the component having detected it,
- not to perceive from the outside an error signaled by a component.

In the events chain that caused the destruction of the first Ariane V launch, this last situation occurred. The two implicated components are the Inertial Reference System (in charge of the determination of the rocket position according to the verticality) and the On-Board Computer (which controls the evolution of the launcher: engines, etc.). A same parameter shared between these two components could be interpreted as flight data or diagnosis data (an error identifier). When the failure of the first computer of the Inertial Reference System was detected, the second computer took over (*Recovery Blocks* technique). This second computer reached rapidly the same erroneous state. This alternate signaled then this error to the On-Board Computer which interpreted this information as a flight data, provoking the rocket swiveling. In this example, the communication of an error not handled, caused an error contamination.

The implementation of tolerance mechanisms requires to explicitly separate the expression of a correct execution of a module and the signaling of an error. Thus, in the case of a program, an error parameter, even distinct from the normal output data returned to the calling routine, can be ignored or misunderstood by this calling routine. On the contrary, the propagation of an exception will interrupt the normal execution control flow by an automatic branching to the exception handler.

18.7 SOME APPLICATION DOMAINS

In this section, we briefly present some fault-tolerant applications, in order to illustrate the techniques introduced in the preceding sections. We also point out that numerous combinations and variations of the basic techniques can be used.

18.7.1 Watchdog and Reset

At first, we consider an industrial electrical power distribution control system. Several control units (boards) are connected through a *CAN Bus*, as shown in *Figure 18.20*. The Master Control (MC) Board, connected to the process, activates a watchdog every 500 ms, sending it a signal. If this operation is not performed, the watchdog provokes a reset of the board (re-initialization of the internal registers, the program counter, the I/O interface and the CAN bus circuitry). This example illustrates a *retry mode* technique applied to the MC board by using a deadline detection mechanism. Very simple, this technique allows escaping from blocking situations due to transient faults (e.g. external perturbations or exceptional conflicts to the Bus access).

This basic mechanism can be extended to other boards of the network which must periodically answer identification requests emitted on the Can Bus by a supervision module. These simple techniques belong to detection and correction mechanisms implemented in this medium-criticality application.

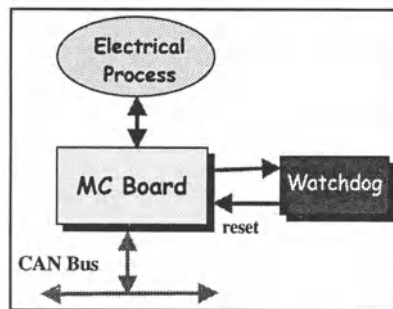


Figure 18.20. Use of a watchdog

18.7.2 Avionics Systems

The electrical flight control system of the Airbus A320 tolerates faults using two techniques called *N-self Checking* and *Double-Duplex*. These fault tolerance approaches jointly use techniques previously introduced.

The specification of the system is used to produce several versions created by different design teams. All the versions are embedded and are executed, but the spares are only used if an error is detected. Each version or group of versions is self-checked, that is it possesses an acceptance test to check the result. Two implementations will be now examined: *N-Self Checking*, and *Double-Duplex*.

18.7.2.1 N-Self Checking

For the *N-Self Checking*, the N versions are executed in parallel, but the N results are not compared. If an error is detected by the acceptance test of the first version (V_1 in *Figure 18.21*), then the result of the version is not accepted. Therefore, the conclusion of the check (AT_2) of the second version (V_2) is considered, etc. So, each version is self-checked and the first correct result is supplied.

This solution combines the advantages of:

- the *N-Versions*, as the actual parallelism of the version computation saves time, particularly when one or several versions run erroneously,
- the *Recovery Blocks*, as the detection of the failed version makes maintenance easier. For instance, the identification of the erroneous hardware subsystem is memorized and this subsystem can immediately be changed after the aircraft landing.

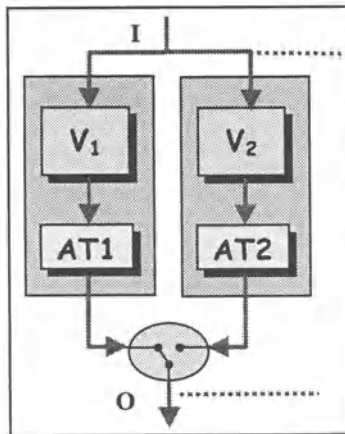


Figure 18.21. N-Self-Checking

18.7.2.2 Double-Duplex

Double-Duplex structure provides another mixing of tolerance techniques. To avoid complex self-checking, couples of versions are considered. The results of 2 versions are compared. If the results of the first

couple are different, they are not considered, and those of the following 2 versions are examined. *Figure 18.22* represents such a structure.

The simultaneous running of the N-Versions saves time. The comparison of the results of 2 versions makes the check of the results easy. The second version may be simpler than the first version and provide an approximated result. Then, the checker examines the coherence between the two results, and not their strict equality. This corresponds to a likelihood technique. The final result is naturally taken from the first version.

The switching to the second couple uses the Recovery Blocks principle. This avoids the implementation of a complex voter required by an N-Versions technique.

Double-Duplex structure has a drawback: we know that at least one of the versions of the couple detected as erroneous fails. However, additional test must be applied to obtain its identity: the first, the second or both?

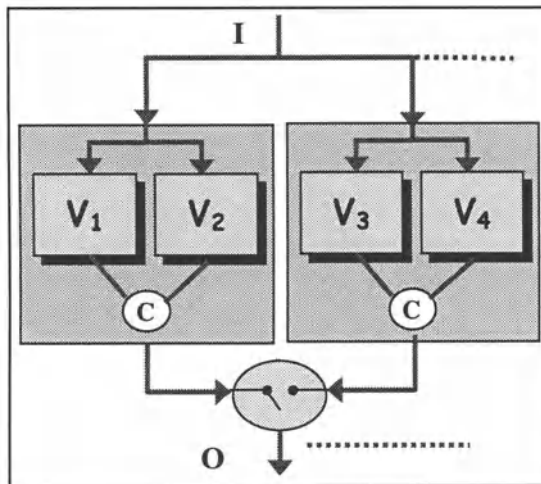


Figure 18.22. Double-Duplex

18.7.3 Data Storage

From the beginning of the Computer Science, detection and correction mechanisms have been introduced in the computers in order to compensate the reliability weaknesses of the technology. This has been especially the case for the main memory, then for the mass memory. IBM began to use error-correcting codes in the IBM 7030 (Stretch) computer with a modified Hamming SEC/DED (Single Error Correcting, Double Error Detecting, code presented in Chapter 15) for main memory. Besides, various redundant techniques have been used in other units of this computer: parity checking for registers, modulo-3 residue check for floating point arithmetic unit,

parity codes for tape recording, and SEC/DEC codes for disk.

Mastering of faults of main memory and mass memory reveals quite different problems, as the used technologies are also quite different. Consequently, the proposed solutions are also original.

18.7.3.1 Main Memory

Nowadays, Random Access Memories use semiconductor integrated circuits (*Static* or *Dynamic* RAMs). These circuits can be affected by two main classes of technological faults:

- *permanent faults* qualified as *hard faults*, such as a ‘stuck-at 1’ fault of a memory cell,
- *temporary faults* qualified as *soft faults*, such as a discharge of a memory cell due to an alpha particle which enters the chip.

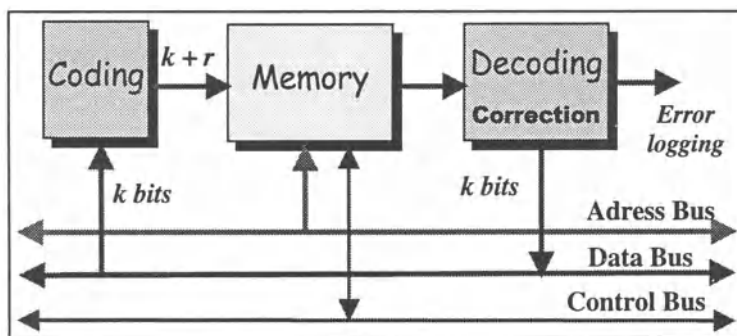


Figure 18.23. Main memory EDC

The use of error detecting and correcting codes like the Hamming modified code to code the data stored in the memory allows the on-line detection and the correction of certain errors. The general structure of such an on-line detection and correction structure is illustrated by Figure 18.23. For example, any single error due to a ‘hard’ or a ‘soft’ fault is corrected. This correction mechanism is based on a *forward recovery* because it does not use the incoming data value (k bit word); the correct value is elaborated from the current output value ($k+r$ bit word). A simple example of such code is proposed in Exercise 18.8.

The differences between ‘hard’ and ‘soft’ fault can be exploited, in order to increase the fault tolerance of this memory. For this purpose, we add a memory management unit (EDC Unit) aiming at coding, decoding, detecting and correcting. Moreover, this unit brings another functionality (Figure 18.24). Besides the classical detection/correction function, each time a word is detected wrong, the unit checks if the detected error results from a ‘hard’

or a 'soft' fault. This is obtained by re-writing in the memory the corrected word: if a new read gives again an error, it implies that it was a hard fault which is signaled at the **error logging** output, else, the word has been corrected.

These **scrubbing** operations are conducted off-line, without any penalty in the normal relations between the memory and the external units (CPU, Direct Memory Access etc.). Specific integrated circuits are proposed by various IC manufacturers to interface the main memory and to manage errors.

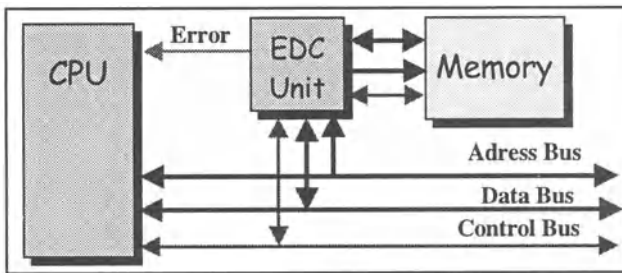


Figure 18.24. Main memory error management unit

18.7.3.2 Mass Memory: Introduction

All mass storage devices, magnetic - optical - or magneto-optical, use redundant EDC codes for on-line correction of certain classes of errors. The faults that can affect such media are not exactly the same as those occurring in main memories. Thus, a new fault class leads to read erroneous words that were however correctly recorded. These random problems are due to the access technology, such as, for example, imprecision of the positioning of the flying heads above the tracks of a magnetic disk. Consequently, in addition to EDC-ECC codes, a retry technique is used: when uncorrectable read errors are detected, the word is read again several consecutive times (for example ten times). We should note that this repetition is a cause of the slowing down of the read access times of some mass memory units having bad quality disks.

Error detecting and correcting mechanisms must be located as close as possible to the disk drive (in the external controller or entirely within the disk drive), in order to eliminate the need for delay in the case of errors and thereby to improve the system performance. Thus, no penalty is put on the communications between the disks and the DMAC (Direct Memory Access Controller).

Independently from the previous EDC code implementation, one can also duplicate storage units on distributed storage systems for high dependability

applications. An example of such redundant approach is the RAID (Redundant Array of Independent Disks) technique, which is presently very much used by many computer system and network manufacturers. Berkeley University has developed this technique which offers several safety levels.

18.7.3.3 CRC for Disc Drive

Disk drive is also subject to errors during read and write operations. The errors can be induced by random noise, correlated noise, media defects, mechanical non-linear phenomena, and other causes. The purpose of error-correction is to improve the integrity and the recoverability of data.

Burst-correcting code are appropriate to master such problems. Codes that are frequently used are the Fire codes which are CRC codes (see Chapter 15).

Example of generator of a Fire code:

$$g(x) = (x^{23} + 1)(x^{12} + x^{10} + x^9 + x^7 + x^6 + x^4 + 1).$$

This code allows the correction of a single 12-bit burst in a sector of the disk (2147 bits). During a write of data to the disk, a division modulo the above polynomial occurs. The remainder is appended to the data and written onto the disk. Hence, 35 bits are added in this coding process.

During a reading, the data stream is again divided modulo the chosen polynomial. If the remainder, or syndrome, is equal to zero, the data is correct. If the syndrome is not equal to zero, then an error has occurred. There are two error types: correctable and uncorrectable. For a correctable error, the syndrome contains information about the error pattern and the error location.

Error detection/correction on Reads

Error detection is critical to ensure the integrity of customer data. The Fire code can detect most uncorrectable errors. But there is a chance that an uncorrectable error will be mistakenly found to be correctable and correction attempted! To prevent this, a 16-bit CRC code is appended to the data on a write before encoding. It is used for error detection after error correction with the Fire code has occurred. This decreases the probability of undetected errors to less than one occurrence in 10^{15} bits transferred.

Moreover, the correction is performed in real-time, during the normal data access operations.

Fault Detection on Writes

A read operation is appended to the write operation, in order to detect a bad recording. This helps preventing a customer from storing bad data without knowing it.

18.7.3.4 CD ROM

Data stored on CD-ROM also implement cyclic codes for the detection and the correction of errors occurring during write and read accesses. The solutions are different in the case of Audio CD and Data CD.

Audio CD ROM

An Audio CD-ROM has 330000 sectors; each sector has 3234 bytes and is structured according to the format of *Table 18.1*. The User Data contains the useful data coding the sounds. The control block stores information concerning the duration and the number of each selection.

The EDC/ECC (Error Detecting Code / Error Correcting Code) part contains two groups of 392 bytes. The *Reed Solomon* code which is a BCH cyclic coding (see Chapter 15) has been first used on the R-DAT (Digital Audio Tape). This code has then been improved as the *Cross Interleaved Reed Solomon Code* (noted CIRC). The code which is used for the compact disk adds some parity symbols and makes an interleaving of the bytes. Thanks to 2 cyclic codes, one before and the other after the interleaving operation, it is possible to detect many faults: thus, 4000 consecutive bits can be recovered and 12 000 bits compensated.

User data	EDC/ECC		Control
	1 st part	2 nd part	
2352	392	392	98

Table 18.1. Audio sector

Data CD ROM

The binary data stored in CD-ROM are structured in a quite similar way. The 3342-byte sectors are divided into three parts, as described in *Table 18.2*. A header and synchronization block is followed by the user data. Then, two 441-byte EDC-ECC blocks integrating the control data are added, and the sector is completed with a third 288-byte EDC/ECC block.

Header Synchronization	User data	EDC/ECC and control	
		parts 1, 2 & control	3 rd part
124	2048	882	288

Table 18.2. Data sector

Comparison

The tables given above show that the redundancy necessary for synchronization, control and EDC-ECC respectively represent 27% of the sector length in Audio CD and 38% in Data CD. The EDC-ECC parts represent redundancies of respectively 32% and 35%.

The two domains do not have the same reliability requirements. A perturbation of one sector of a 74-minute audio CD-ROM will affect 13 millisecond of the produced sound. This will probably have no consequence on the user. On the contrary, any loss of a byte in a data file of a data CD-ROM can have severe consequences on the use of a file that is read (executable program, parameters, etc.).

In fact, the reliability of actual Data CD-ROM units (media, read and write operation, transfers) is high: we find industrial products with announced values of 10^{-12} errors per CD (of 650 Mbytes).

18.7.4 Data Transmission

Data transmission makes use of classical EDC codes since the first ages of this science. Thus the classical protocols, such as SDLC, HDLC, X25 use cyclic codes. For example, the following standard polynomial code is used for signature in disk controllers and data communication protocols:

$$g(x) = x^{15} + x^{12} + x^5 + 1.$$

The redundant codes are not often used in the classical data Bus of computer systems. Frequently, only simple parity bits are used, which does not allow error correction. On the contrary, in high-critical applications, more sophisticated codes are implemented.

We will describe the tolerance mechanisms used by some significant LAN industrial networks which does not belong to high-critical applications: the Ethernet Bus, used in general purpose LAN network (commercial or scientific networks), and the CAN and VAN Busses, more recent, which belong to Industrial LAN dedicated to real-time applications.

18.7.4.1 Ethernet Network

Two error checks are made at the ISO level named 'control':

- detection of erroneous bits on the physical media,
- detection and recovery of message collisions.

The complete error handling is relegated to higher levels of ISO architecture.

According to the *security* criterion (*confidentiality* and *integrity* parameters), we should note that no encryption of the transmitted data is made.

Alarm Thresholds

Several parameters are used to detect the occurrence of errors in the global functioning of the network.

First, the maximum *workload* on the network is evaluated and compared to a maximum value. It is estimated that a network is overloaded when its average use is higher than 25%. This figure corresponds roughly to traffic peaks of 60%. Above, it is necessary to re-route a part of the traffic activity. This threshold corresponds to:

- 750 000 bytes per second for a 10Mbits/sec with a load of 60%, as the Ethernet network traffic is $(10\text{Mbits} / 8\text{bits}) \times 0,6 = 750\ 000$ bytes per second;
- 3000 frames per second because, for an average frame length of 250 bytes, the network transmits $750\ 000 / 250 = 3\ 000$ frames per second.

A network is considered as reliable if its error rate does not exceed 0,01 %. At full admissible load, 3 000 frames per second being transmitted, this rate corresponds to an average of 0,3 error / second, thus 1 080 errors / hour. By experience, we admit that there are 3 brief error peaks per hour. Hence, these peaks must not exceed $1\ 080/3 = 360$, which justifies a guarded peak limit at 300. The maximum value is then of 300 errors / second.

The workload is a first parameter used to detect errors; the *message collision number* is a second one. The maximum number of message collisions on the Ethernet Bus is estimated at 50. This result corresponds to an average using, with peaks of 35 to 40%.

Finally, the *broadcast frames* are special frames send to all the stations connected to the network. The average number of broadcasts per second is about 0.3. We can use this threshold to separate possible broadcast storms from the normal background noise. These storms are frequent under TCP/IP, due to a bad interpretation of a local address. Unfortunately, they can paralyze the network: a broadcast is treated by all the stations, resulting in a saturation of the equipment. Thus, the reaching of the threshold must be detected as error.

Error Model

The detection of anomalies leads to *error logging*. The erroneous frames are stored in memory. These errors are, for example:

- too short or too long frame,

- wrong CRC (redundant EDC-ECC),
- bad alignment of the data.

The 20 to 63 byte frames are considered as erroneous frames that do not result from message collisions. Indeed, a collision produces a non-significant frame whose length is generally between 5 and 20 bytes.

The too long frames correspond to a length greater than 1512 bytes.

Finally, errors on the transmitted data are detected if they alter the CRC, or if they produce a bad alignment (wrong positioning).

The addresses of the erroneous frames are also recorded, in order to diagnose the source of the problem (thanks to the identity of the emitter).

Diagnosis and recovery

The detected errors lead to a diagnosis and a recovery treatment. Here are some examples:

- Short frames without significance: they are characteristic of collisions which are tolerated thanks to the re-emission technique (retry mode).
- Erroneous frames coming from the same station: the station is declared as failing, leading to a maintenance procedure.
- Intense traffic at certain times during the day: this is due to peaks of use, leading to re-routing of some transactions.
- Intense traffic between two distant stations: they must be moved and connected to a same network segment (reconfiguration).
- broadcast traffic over the pre-defined threshold. The destination address of such frames is FFFFFFFFFFFFFFFF (12 'F'). A broadcast overflow can come from a failure of the emitter of a station (stuck-at 1 type), or of one application. Under TCP/IP, this can be a station that sends a wrong broadcast by emitting the local address '0' instead of the correct 'F' one. All the other stations will try to make this frame follow, as it should normally be done; as they do not recognize the address, they send a broadcast message. Hence a saturation occurs.

18.7.4.2 CAN Bus

Initially developed in Europe for automotive applications, the CAN (Controller Area Network) protocol is now used in more general industrial *loosely coupled systems* using control devices, sensors and actuators. It has been standardized under ISO 11898, and many micro-controllers now implement a CAN interface.

The CAN Bus is a two wire serial data bus able to transmit random asynchronous information at up to 1 Mbit/s. It follows the ISO protocol

standard and implements the levels 1, 2 and 7. Each frame contains a Start bit, an identifier field for address and priority of the message (11 bits in CAN version 2.0 A, and 29 bits in CAN version 2.0 B), a RTR (Remote Transmission Request) bit, a 6-bit control field, a data field of 0 to 8 bytes, a 15-bit CRC + 1-bit delimiter (at high value), a 2-bit ACK field (including an ACK delimiter at high value), and a 10-bit EOF + Inter Frame Space. The bit stream is coded according to the NRZ technique (Non-Return to Zero). The electrical level 'low' is dominant while the level 'high' is recessive. This means that if two nodes emit a 'high' and a 'low' levels at the same time, the 'low' level is transmitted (wired AND). The *bit stuffing* technique is used: after the 5th bit with equal polarity, a 6th additional bit with opposite polarity is stuffed into the bit stream. The CRC delimiter and the ACK and EOF fields have a fixed form and are not stuffed. When the Bus is not used, the line is idle (level high). Any active node of the system is allowed to start a message transfer. The CAN Bus is a multi-master with a CSMA/CD + AMP (Carrier Sense Multiple Access with Collision Detection and Arbitration on Message Priority) Bus access method. Each node, which emits a message, reads back the transmitted bits. As soon as it detects a collision (according to the wired AND principle), it stops the emission and switches to the reception mode. It will try again later when the Bus is free: this is a delayed *retry mode*.

A CRC checksum is used for error detection only. It is possible to detect up to 6 single bit errors or up to 15 bits burst errors.

The CAN controller contains an error management unit, which handles errors. Each time an error is detected by a node, it will be immediately noticed to the remaining part of the network by an *error frame*. After this error message, all nodes discard the received bits and the emitter will repeat its message later (*retry mode* technique). This activity is managed at low level, by the CAN controllers.

The error process comprises 2 main steps, *error detection*, and *error handling*, which are examined in the following paragraphs.

Error detection

Five different errors are detected on the CAN Bus:

- bit error: when the value which is monitored on the Bus is different from the bit transmitted, according to the basic signal coding,
- bit stuffing error: when 6 consecutive bits have the same value,
- CRC error: when an error is detected by the cyclic code checking,
- form error: when a fixed form field (CRC delimiter, ACK, EOF) contains illegal bits,

- acknowledgment error: each active node that detects a correct message on the Bus overwrites the recessive (high) delimiter of the ACK field with a low level; if the transmitter does not monitor a dominant bit (low) during the ACK field, it detects an acknowledgment error.

Error handling

When a CAN controller detects an error, it notifies this error to the network by an *active error frame* which uses a violation coding. This error frame contains a flag which is made of 6 or more 'dominant bits' realizing a bit stuffing violation or destroys a bit field in fixed form. This violation is detected by all the nodes which send error frames. In order to avoid error contamination, an algorithm is used to disconnect the defective nodes and restore the correct Bus access.

Two kinds of error frames, *active* (dominant) and *passive* (recessive), are used on the Bus, and each node has three states according to *Figure 18.25*:

1. *Error active*: the node can communicate on the Bus, and it sends an active error flag when an error is detected;
2. *Error passive*: the node can communicate on the Bus, and it sends a passive error flag when an error is detected;
3. *Bus off*: this state is a fault confinement state in which the node cannot send or receive any message.

Two counters, *Transmit Error Count* and *Receive Error Count*, are used by each node to evolve between the normal state 'error active' and 'error passive' state, and then possibly in Bus off state (which is a trap state which need a reset operation to return to the error active state).

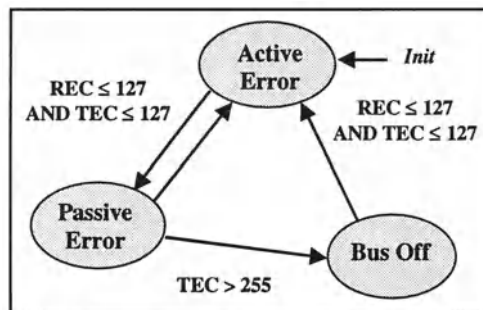


Figure 18.25. State graph of error handling

18.7.4.3 VAN Bus

The *VAN (Vehicle Area Network)* Bus is another random asynchronous serial Bus used in Automotive industry and which offers interesting

protective mechanisms. The VAN protocol normalizes the levels 1 and 2 of ISO model (reference ISO 11519). The medium uses 4 wires (two Data wires, Ground and Vcc), and the Data are transmitted according to a differential current mode for a better quality of service: better resistance to electromagnetic perturbation, and possibility to continue the transmission if one Data wire is cut or is in short-circuit to Ground or Vcc. The signal is coded with a *E-Manchester* coding which is a mix of NRZ and Manchester introduced in Chapter 15.

The arbitration is based on a non-destructive CSMA/CD technique on all the frames of the messages. A 15-bit CRC is used for error detection and correction. The generator is:

$$\begin{aligned} g(x) &= x^{15} + x^{11} + x^{10} + x^9 + x^8 + x^7 + x^4 + x^3 + x^2 + 1 \\ &= (x^7 + 1)(x^8 + x^4 + x^3 + x^2 + x + 1) \end{aligned}$$

This Fire code is able to detect isolated errors, burst errors of length lower than 15 bits and double burst errors of length lower than 8 bits.

18.8 EXERCISES

Exercise 18.1. Reliability of the TMR

Calculate the reliability of a TMR described in section 18.2, for exponential reliability laws with constant failure rates:

1. If we neglect the reliability of the voter;
2. If we suppose that the failure rate of the voter is ten times smaller than the failure rate of the duplicated modules.

Exercise 18.2. Fault tolerance of the TMR

Analyze the TMR structure and determine single and multiple faults which are tolerated and those which are not tolerated.

Criticize this structure.

Exercise 18.3. NMR

We want to make the logical design of the detection and correction module of a NMR structure.

1. Verify the logical expression of the error detection signal proposed in sub-section 18.5.3.
2. Find the logical function having three inputs ($S1$, $S2$ and $S3$) and three outputs ($M1$, $M2$ and $M3$), and which identifies the failing module.

3. Determine the logical function of a 3-input voter, then a 4-input voter. What use would be a 4-input voter?

Exercise 18.4. Study of the double duplex

Detail the analysis made in sub-section 18.7.2.2. Calculate the Double-Duplex reliability.

Exercise 18.5. Study of self-purging technique

Each one of the n modules of a self-purging system compares its output with the final output of the system. In case of difference, it switches itself off, and the vote is continued with $(n-1)$ modules.

What is the interest of such technique?

Imagine its implementation as several software tasks distributed into a computer system.

Which properties still has this system when two modules only remain active?

This question can also be applied to the *double duplex* redundant structure of Exercise 18.4.

Exercise 18.6. Example of a tolerant program based on retry mode

A procedure `Get(I)` reads the characters which are pressed on a keyboard until the 'return' key has been keypressed. These characters are then converted into a decimal value which is assigned to the variable I . If characters different from the figures are provided, the execution of the procedure is suspended and the exception `Data_Error` is raised.

Define a procedure `Safe_Get(I)` able to tolerate the keypressing faults.

Exercise 18.7. Programming and evaluation of recovery blocks

1. *Programming.* Let us consider a function P having one parameter C of type T , and returning a Boolean value signaling an error. Propose two implementations of a procedure tolerating the faults of P , thanks to an alternate Q , according to the two approaches of the recovery block technique examined in section 18.4.2.
2. *Performance.* Study the temporal performance of these two solutions.

Exercise 18.8. EDC in a RAM

We consider a Memory Management Unit such as the one introduced in sub-section 18.7.3.1. The data words have $k = 8$ bits, and they are coded with a linear code using $n = 12$ bits (hence $r = 4$). The coding relations are the following:

$$y_1 = u_1 \oplus u_2 \oplus u_4 \oplus u_5 \oplus u_7,$$

$$y_2 = u_1 \oplus u_3 \oplus u_4 \oplus u_6 \oplus u_7,$$

$$y_3 = u_1,$$

$$y_4 = u_2 \oplus u_3 \oplus u_4 \oplus u_8,$$

$$y_5 = u_2, y_6 = u_3, y_7 = u_4,$$

$$y_8 = u_5 \oplus u_6 \oplus u_7 \oplus u_8,$$

$$y_9 = u_5, y_{10} = u_6, y_{11} = u_7, y_{12} = u_8.$$

1. Draw the generator and the control matrices, G and H (see Chapter 15). Study the coding and decoding of the 8-bit word: (00111011). Determine the Hamming distance between all codewords.
2. Analyze the influence of a single error on the previous word. How to correct this error?
3. Discuss the implementation of this code in the MMU.
4. Is this code appropriate for *scrubbing* 'soft faults'?

Chapter 19

Conclusions

In this book, we have, for pedagogical reasons, introduced problems and solutions in a progressive way, providing separated viewpoints on the design of dependable computing systems. In this concluding chapter, with the same pedagogical objectives, we will first make a synthesis of the numerous aspects we have encountered. This synthetic view is now necessary to reorder the most important notions explained in the preceding chapters. Section 19.1 summarizes the needs and the impairments which are at the origin of studies on dependability. In section 19.2, we consider the three classes of protective means introduced to obtain dependable products: *fault prevention*, *fault removal*, and *fault tolerance*. For each of these classes, we have studied numerous techniques. The question of their respective efficiency is now addressed. These techniques aim at increasing the dependability, that is the reliance that can justifiably be placed on the service delivered. Hence, we must evaluate them using the dependability attributes and *assessment* techniques discussed in section 19.3. We analyze here only three attributes of the *quantitative approaches*: the redundancy which deals with the structure of the product and/or its intermediate models, the reliability and the safety. An overview of the *qualitative approaches* is then provided.

Finally, we conclude in section 19.4, with a brief analysis of the difficulty of choosing adapted dependability means, but also a discussion about the real (undeniable) contributions of dependability techniques.

From the basic knowledge provided in this book, the reader is now able to go deeper into some aspects of the large domain dealing with the design of dependable computing systems.

19.1 NEEDS AND IMPAIRMENTS

19.1.1 Dependability Needs

One of the main evolutions of computing systems is the *increase of the responsibility delegated* to them. Thirty years ago, their role was limited to simple services explicitly controlled by users. These systems were employed:

- to increase the human activity productivity such as numerical computation allowing simulation results to be quickly achieved,
- to improve everyday life such as electronic ignition systems, reducing car pollution.

Then, many systems became human assistants. For instance, an ABS (Anti-Blocking System) prevents the car wheel blocking, taking the reactions of the driver as well as the environmental conditions into account at braking time. However, the driver sends an order to the system, pressing the braking pedal. Nowadays, computing systems are substituting for human to take themselves decisions. The airbag opening in a car is controlled by such a system.

The *increase of computing system complexity* is a second characteristic of the advancement of these systems. This increase is due to a greater number of provided services (quantitative complexity), to complicated algorithms (qualitative complexity) and to the increased interactions between the constituent subsystems. Once again, computing systems embedded in cars provide good examples. Their number increases: systems for fuel injection, ignition of the sparks, management of braking, steering and stability of cars, air conditioner regulation, etc. The implemented control laws are more and more complex. For instance, the engine control algorithms are more sophisticated to decrease pollution (ignition advance, feedback of data characterizing the non-burned gas composition, etc.). These systems become more and more highly coupled. For example, to control a car going too fast into a bend, the engine control systems and braking management systems must handle complex interactions.

The two quoted characteristics, that is, increase of the responsibility and increase of the complexity lead to a contradiction. On the one hand, the increasing complexity of the systems makes inevitable the rise of the faulty design risk and, consequently, of system failure risk during operation. On the other hand, more and more responsibilities being delegated to these systems, the occurrences of such failures is more and more unacceptable.

This contradiction leads the computing system users to ask for dependable systems. This demand is justified by numerous tragic issues,

including patients killed by the failure of medical equipment, drivers injured in a car accident due to untimely airbag opening, the loss of the first launching of Ariane V.

Dependability requirement has been extended to non-critical systems. For years, computing system failures were accepted by users as inevitable. Nowadays, economical constraints imposed on users of computing systems (service quality, hard deadlines, etc.) make them less lenient.

19.1.2 Dependability Impairments

The book aims at defining the basic concepts associated with the dependability domain. Setting the problem to be solved, structuring the requirements at the origin of this problem and the means to solve it are essential aspects. They constitute the basis allowing dependability to be considered as a science and not as an aggregate of experimental techniques used by engineers.

Faults, Errors, Failures, Consequences

This foundational work at first took up studies on dependability *impairments*. Four essential notions were defined:

- *Failure* characterizes a wrong service delivered by the computing system. The product has an actual behavior that is not in compliance with the expected behavior, as defined by the specification. This notion concerns the product considered as a black box.
- *Fault* is a failure cause. It is often expressed as a non-respect for a property on the designed system structure. A connection of an integrated circuit being broken, or a wrong program statement, are two examples. Thus, this notion concerns the structure of the product, that is a system defined as assembled components, and more precisely a static view of this structure. To point out the failure origin is sometimes difficult, for instance when detailed knowledge on the structure is not available, or when the causes come from outside or are multiple and combined.
- *Error* is an intermediate notion. It characterizes the fault effect as an undesirable internal functioning state of the system behavior. A gate output stuck-at 1 or the access to an array element that is out of range are two examples. Therefore, this notion concerns the dynamics of the system.
- *Consequence* is an external notion. It defines the effects of the failures on the environment (user and non-functional environment), and on the product itself, as it may be destroyed.

A cause and effect relationship exists between fault, error, failure, and consequence. However, all faults do not mandatory lead to an error, which does not necessarily provoke a failure. For example, a bad statement of a program (that is, a fault) may have no effect (no error produced) if this statement is not executed (particular use of the program, or dead code due to reuse). The assignment of a value in an array, out of range, is an error. It does not provoke a failure if the assigned address is in the memory data segment and if the crushed value is no longer used by the program execution. In the same way, a failure causes more or less perceptible or acceptable consequences.

Modeling Tools

As in other sciences, dependability looks for modeling tools of the handled concepts: faults, errors, failures and their consequences. Models then allow generic solutions to be deduced, that is, solutions applicable to a class of problems or systems and not limited to one specific problem concerning one system. These modeling tools also allow assessing the efficiency of the means proposed to handle the impairments. For instance, the test method relevance depends on its capability to detect the presence of faults. As actual faults existing in a system are generally unknown, fault models are often used to evaluate the test techniques.

The possible faults in a computing system are innumerable. They depend on the functional characteristics of the system, on the used design modeling tool, on the design process, on the implementation technology, on the user of the system and on the non-functional environment (temperature, radiation, etc.). The proposal of only one model of faults, errors or failures is not realistic. On the other hand, the handling of each specific fault of each system requires empirical and expensive studies. So, numerous modeling tools for faults, errors, failures, or consequences were proposed. Modeling tools being generic, that is, independent of specific characteristics of particular systems, they led to a scientific approach of dependability studies.

A first set of modeling tools aims at characterizing the failures and their effects. The considered modeling criteria concern human, economical or environmental damages. The *seriousness of the consequences* of the perturbations caused by the failures is assessed. The modeling tool based on the following classes is a conventional example: benign, significant, serious and catastrophic. It is a model, that is, an abstraction of the term *failure*; fault models provide other abstractions on dependability impairments, which are independent from this first abstraction. For instance, a given hardware fault in a micro-controller may have quite different consequences depending on whether this circuit is used in a game station or integrated in an embedded flight control system. Seriousness is one model among many characterizing

failures. For example, *inertia*, that is, duration between failure occurrence and its consequences, or *failure risk*, that is, the occurrence probability of a failure, are other criteria leading to other classes characterizing the system dependability. This example shows that various modeling tools must be used to express dependability impairment concepts. Each modeling tool providing one point of view, several of them are necessary to characterize faults, errors, failures and consequences.

The study of system faults and of means to handle them requires the choice of *fault modeling tools*. A fault-modeling tool is defined by properties on the system structure model. For instance, the programming language syntax definition implicitly expresses a fault modeling tool: each violation of a syntax rule by a program text is a fault. Of course, the fault modeling tools depend on the modeling means used to express the system structure. However, several fault models may be proposed for one system modeling means. To define a fault modeling tool, the expected or unexpected properties must be generic, that is, independent of specific systems. For instance, a particular statement of a given program, which is not in accordance with a rule of the programming language syntax, contains a fault specific to this program. Nevertheless, the syntactic rules, which define a fault-modeling tool, are applicable to any program.

Fault modeling tools have numerous applications, such as efficiency measurement of fault detection techniques used to extract faults (fault removal) or to tolerate activated faults (fault tolerance). Unfortunately, the exclusive use of these models is not sufficient. In particular, numerous actual faults cannot be expressed with these models. They do not offer a full coverage. Such a situation exists for instance to study software design faults. Consequently, the faults are also studied from their effects on the internal system functioning. Bad states are defined by *error modeling tools*. Once again, numerous error modeling tools have been proposed. Some of them are general, that is, they do not depend on the system modeling means. For instance, characterization of errors as *permanent* or *temporary* defines two classes. Other models are derived from the semantics of the modeling means. The use of the value of a variable not previously assigned is an example of an error model associated with the behavioral model defined by the programming language semantics. The run-time stack overflow defines another property violation, that is, an error model, associated with the object code model. A deadlock, that is, a system state progress blocking due to internal interactions of subsystems is another error model for systems modeled by Petri nets or concurrent tasks.

The assessment of *the relevance of the fault or error modeling tools* is a difficult issue. The answer often depends on the use of the models. For instance, the program test technique evaluation method based on mutations

considers the replacement of an arithmetic operation by another as a fault-modeling tool. Of course, no engineers probably do such a fault. He/she makes more complex wrong structure modifications. However, these mutations produce errors characteristic of behavioral effects of actual design faults. Thus, these fault models are successfully used to assess functional test sequences. Besides relevance, these modeling tools must be examined in term of tractability, that is, their capabilities to be processed. For instance, a fault simulation with a too precise model may take prohibitive duration.

Generic fault or error modeling tools do not allow to express numerous faults or errors which are specific to the structure or the functioning of each system. Therefore, specific faults or errors must be expressed. However, to preserve the capability to do generic studies on them, macro-models (or macro-languages) are proposed to express them.

19.2 PROTECTIVE MEANS

To improve the computing system dependability, three approaches are proposed:

- *Fault prevention* aiming at reducing the creation or occurrence of faults during the system life cycle.
- *Fault removal* aiming at detecting and eliminating existing faults, or at showing the absence of faults.
- *Fault tolerance* aiming at guaranteeing the correctness of the services delivered by the system despite the presence or appearance of faults.

19.2.1 Fault Prevention

Fault prevention aims at reducing the creation or occurrence of faults during the computing system life cycle. Several techniques can be used during the system design phase. Some of them have an impact on the created system. Others prevent faults occurring during its future useful life (operational phase). These means concern the system modeling tools (including implementation technologies), the system models and the processes used to obtain these models. These three viewpoints are developed hereafter.

The *modeling means* has an important effect on dependability of the modeled systems. This fact is well known for implementation modeling tools. Certain technologies are safer than others. For instance, they prevent faults introduced during the hardware system manufacturing step or occurring during the operational phase in hostile environment (space for

example). Some investigations concern the software implementation modeling tools. Studies on requirement, specification and design modeling means, using their capability to prevent faults as analysis criterion, is relatively little-developed.

A modeling means being selected, numerous *modeling choices*, and therefore numerous *system models*, exist for one particular product. Generally, the choice is guided by performance or maintainability criteria but rarely looking for fault prevention. At first, the obtained models aim at being operational, that is executable. Due to this single goal, the pieces of information coming from the system origin (Why this system is useful? What system must be designed?) are lost: they are not present in the created model. They are substituted for data associated with the realization, answering to question: How the system is designed? The preservation of the first type of pieces of information in the models, certainly not useful for operation, is an efficient means to prevent faults. This example illustrates the general concept of *redundancy*, also used for fault removal and fault tolerance purposes. Whereas the term ‘redundancy’ is conventionally regarded as the meaning ‘supplementary’ and ‘not useful’, we showed that it is an efficient concept to obtain dependable computing systems.

Mastering the *model creation process* is a third means of fault prevention. This point is correlated to *quality* policies, processes and procedures. They aim at modeling the development process, at assessing its efficiency to prevent faults and at improving the process. This approach is original as it concerns neither the developed system, nor the means used (modeling tools, etc.). It studies the human process used to create a system. The underlying idea is that numerous design faults existing in systems are coming from faults occurring in the design process. To avoid the introduction of these faults, guidelines may be provided to master the design process phases. Some of them were given to illustrate this approach on the requirement, specification, design, and implementation stages.

Relationships exist between studies concerning tools, models and modeling process. For instance, the intellectual capabilities of humans being limited, the designers cannot simultaneously handle numerous concepts (process issue). So, the modeling tool must allow abstraction hierarchies to be expressed, offering suitable features such as modules (modeling tool features).

19.2.2 Fault Removal

Fault removal aims at detecting and eliminating existing faults, or at showing the absence of faults. Studies on fault removal are older than those on fault prevention. They were initially justified by manufacturing checks of

hardware systems. Then, they were extended to model checking. The proposed techniques are therefore numerous and varied. We do not want to expose them again one by one. We synthesize hereafter their presentation, introducing criteria allowing them to be classified.

Firstly, fault presence may be detected by *static analysis* (for instance, inspection or property checking) or *dynamic analysis* (for instance, testing). The first class of techniques does not need system execution which is required by the second group.

Secondly, the examined model concerns the system realization (*structural approach*), the function provided by the system (*functional approach*) or both (*structural-functional approach*). In the first case, the presence of faults is highlighted; in the second case, the studies handle failures; in the third case, error notion is the main concept as well as relationships between faults, errors and failures.

Thirdly, fault detection turns on a *certainty* (for example, a specific fault, error or failure) or a *risk*. In this second situation, the techniques search for the potentiality of the presence of fault, error or failure (presence probability). Measurements on systems are defined and associated with risk levels (cf. section 19.3). For instance, the more complex the control flow of a program is, the more difficult is the mastering of the program by its designer and so, the greater is the risk of a design fault being present. The measurements assess the model (complexity measurements) or the use of the modeling tool features. For instance, the use of the 'goto' statement or of shared variables increases the fault risk. On the contrary, the use of an 'else null' option if no action is necessary when a test is negative, decreases the risk of an omission fault.

Detection generally requires a reference model which is compared with the created model. This reference model concerns system requirements or specifications, but also system design or implementation. This model expresses what is expected (approach based on *good functioning*) or unwanted (approach based on *bad functioning*). The reference model is given by extension (everything which is expected or not) or by intention (properties). The reference model may be the one used by the current step (for instance, the specification model at design step) or another. Certain techniques do not need additional reference model. Faults, errors or failures are detected by examining the current system model. Taking the programming model as example, data flow analysis techniques show variables assigned two times without being used, that is, an error.

Most of the techniques handle one of the three notions (fault, error or failure). Some aim at handling links between them. *Fault diagnosis* methods provide such an example. Faults at the origin of an occurred failure, or a detected error are searched.

From the origin, fault removal techniques are used to detect the presence of faults. They are also useful to prove the absence of faults. Then, they also provide means to assess dependability of computing systems.

Finally, assessment of fault removal techniques is another large issue. What are the interests and limits of these techniques? What are their specific contribution and so their complementarity? The answers to these questions are essential to define *fault removal policies*. This knowledge has an immediate practical interest, as economical constraints do not allow all the various offered techniques to be used jointly.

Fault removal techniques are often considered at the end of the model definition, particularly when an operational model of the system is completed. However, these means may have a great influence on the system model or on the modeling process. For instance, we showed how error detection mechanisms can be introduced at design time to make on-line detection easier at run-time.

Fault prevention and fault removal domains are put together under the term *Fault avoidance*. Studies associating these two domains are useful. For example, the definition of a development process aiming at preventing faults often uses fault removal techniques at each process step.

19.2.3 Fault Tolerance

Fault tolerance aims at guaranteeing the services delivered by the system despite the presence or appearance of faults.

Fault tolerance approaches are divided into two classes:

- *compensation* techniques for which the structural redundancy of the system masks the fault presence and,
- *error detection and recovery* techniques, that is, error detection and then resumption of the execution from a safe state previously reached and stored (*backward recovery*) or not (*forward recovery*), and/or after an operational structure modification (*reconfiguration*).

In numerous industrial systems, intensive use of fault tolerance techniques is too expensive, both in terms of development and resources. In particular, this last expenditure is added to each produced system if a fault-tolerant hardware platform is developed. To reduce these over-costs, some benign or at least not dangerous or catastrophic failures are accepted. For instance, when a car quickly overtakes another car, the processor is full-time used to control the engine; then, tasks managing the air-conditioner are temporary suspended. *Fail-safe* techniques jointly study failure seriousness and fault tolerance.

Assessment of the efficiency of fault tolerance techniques is another important problem. It uses fault and error models and again poses the problem of the pertinence of such models. These faults or errors are injected and the system operation is simulated, or their effects on the product behavior are assessed (cf. next section).

Fault tolerance concerns the system for architectural viewpoint as well as software and hardware implementation technologies. The projection of the results of studies done on system modeling onto technological levels is not so easy. This issue comes from the fact that hypotheses often implicitly used at system level are not correct when technology characteristics are considered. For instance, the switch from a failed component to a duplicate is efficient to tolerate faults of hardware technologies. This technique is inefficient to tolerate design faults or most of the software technology faults. Moreover, most of the assumptions associated with a fault tolerance technique are in accordance with one technology, whereas complex interactions exist in real systems between hardware and software elements.

The various approaches used to obtain a dependable computing system (fault prevention, removal and tolerance) were introduced separately. They provide complementary contributions. However, we signaled that they are also correlated. For example, fault tolerance mechanisms aim at increasing the system reliability. However, as these means are often complex, they increase the fault risk and thus they go against fault prevention requirements. Moreover, they can make fault detection more difficult by reducing the system observability. Thus, fault tolerance techniques may lead to a system reliability decrease.

19.3 DEPENDABILITY ASSESSMENT

Whatever the protective means used to avoid failure occurrences, the reliance placed on a computer system must be justified. The measurement of this reliance (quantitative approach) or an evaluation of the presence or appearance of faults and their effects (qualitative approach) provides this justification.

19.3.1 Quantitative Approaches

Dependability, that is, the justified reliance placed on the services delivered by a system, must be assessed to be justified. Numerous attributes exist, taking various meaning for the term 'reliance on the services'. For most of them, a measurement is defined as a conditional probability, which is a function of time.

Attributes

Reliability is one of these attributes. It defines the aptitude to accomplish a required function in given conditions. Thus, reliability measurement is a function expressing the probability that the system has survived without failure at time t , given that it was operational at time '0'. Availability, safety, integrity are other introduced attributes.

Mathematical models describe probability laws of the correct operation of system components. Parameters of these laws frequently depend on the used technology. Their values are coming from experimental facts. Other parameters influence these laws, such as the temperature for hardware technologies.

As a system is a structure of components whose probabilistic models are known, composition operations provide laws for the complete system.

Non-probabilistic models also exist. For instance, some of them are based on the measure of the structural or functional *complexity* of the system: the number of statements in program blocks, the number of paths in the control flow of a program or in a behavioral model, etc. Since humans (the system designers) have limited intellectual abilities, they cannot master highly complex systems. A *fault presence risk* is therefore defined from the designed system complexity measurement.

Redundancy

Redundancy is another criteria useful to evaluate a product and the different protective mechanisms used to improve the dependability. It measures the extra-cost implied by the use of the chosen technique, in terms of quantity of resources (number of electronic components, of functions, etc.), but also its effects on the system dependability.

In *Figure 19.1* seven groups of techniques analyzed in this book are summarized and ordered vertically according to the importance of the implied redundancy. This figure symbolically shows the main life cycle steps involved by each group of techniques (specification/design, production, and operation). We observe that the redundancy of these groups vary from a magnitude '1' (no redundancy) for most of the functional fault avoidance techniques, to a magnitude '2' (100% redundancy) for self-testing techniques, and finally to a magnitude '3' (200% redundancy) for fault tolerance techniques by compensation (e.g. 3-Versions). In fact, these figures are purely symbolic; redundancy figures of real dependable product are frequently much greater:

- integration of redundant specification elements in the design model in order to prevent or detect faults,
- fault-tolerant design techniques using quadruple duplicate modules

(Double-Duplex) for avionics control systems, or else using quintuple duplicate modules for spatial applications.

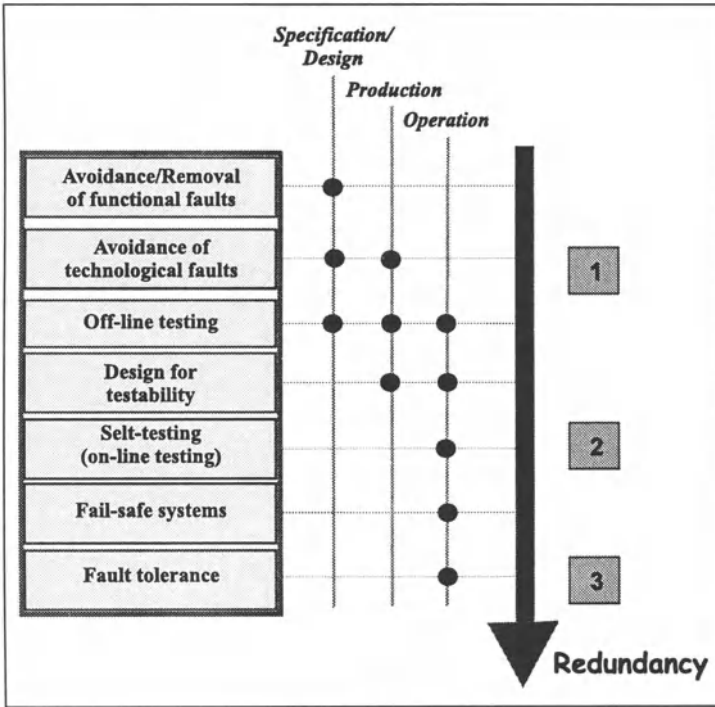


Figure 19.1. Dependability techniques and Redundancy

Let us note that one must not confuse redundancy with financial cost of the product. Indeed, although globally related to the dependability level required, the financial cost of a project is not at all proportional to the final product's structural redundancy. This cost also depends on human and tool means implied by the chosen techniques. Unfortunately, this cost increases very fast, in a non-linear way, along our symbolic vertical redundancy axe. In other respects, the manufacturing processes, which lead to highly reliable components, are very expensive. Consequently, the cost of a product integrating highly reliable integrated circuits is high, even if no redundancy is involved in this product.

Conversely, we insisted several times on the prohibitive cost of selling failing products with the reason of making financial savings (of not using dependability methods). Not only the products but also the credibility of the marks and the manufacturers is questioned by such approach.

The use of techniques based on redundancy is frequently criticized because it increases the cost of each resulting product. This sentence is true

if structural redundancy is introduced in the final implementation of the product (for example, the use of redundant modules). On the contrary, it is false if the involved redundancy is used in the development steps in order to prevent or to remove faults. For example, consider the following sentence «the use of the Ada language is inefficient because the redundant information imposed by its programming features make the generated code heavy ». It is true that Ada imposes redundant elements. However the sentence is false. Indeed, the redundant information allows verification operations at compile time, which detect numerous design faults; moreover, most of the redundant elements do not induce redundant object code.

In the same way, design guidelines may introduce redundancy in the design models without any redundancy in the operational models. For instance, the writing

```
    if Condition then Action;
                                else null ;
    end if ;
instead of
    if Condition then Action;
    end if ;
```

is obviously redundant. However, it avoids any fault omission of the else branch. Moreover, the code optimizers included in classical Ada compilers do not produce any supplementary object code.

Reliability and Safety

The attributes used to assess dependability are sometimes antagonistic. If we consider *reliability* and *safety* criteria only, we can note a potential conflict between groups of dependability techniques of *Figure 19.1*. According to the hypotheses generally made for hardware technology, reliability is inversely proportional to the complexity expressed in terms of number of elementary components. For example, with an exponential law with constant failure rate, this failure rate is doubled when the number of electronic components doubles; thus the MTBF or MTTF is divided by two. Consequently, some techniques used for high reliability design try to reduce the total number of components used. On the contrary, many safety techniques try to detect and correct the errors, or to mask these errors; this is accomplished thanks to the use of redundant codes which adds more components. As a consequence, the resulting products have a higher probability of fault occurrence. From this analysis one can deduce the assessment:

« more safety = less reliability »

The use of redundancy allows to create a safe product, as demanded by the dependability requirements. However, this can lead to a final product which is frequently performing error detection and reconfiguration procedures. As a consequence, this can provoke a degradation of the normal activity of the application (reduction of the performance), or even, to timing failures.

19.3.2 Qualitative Approaches

Qualitative approaches examine relationships between faults, errors and failures. The proposed qualitative assessment methods are distributed in the two following classes.

- *Deductive approaches* consist in deducing potential failures from the system faults or errors for instance specified by modeling tools. They use structural and/or behavioral models of the system to process this deduction. In this book, we have mainly introduced the FMEA (Failure Modes and Effects Analysis) technique.
- *Inductive approaches* consider unwanted failures and infer errors or events, which may lead to these failures, or show that such failures cannot occur. For instance, proof of properties, or FTM (Fault tree Method) are tools to implement these approaches.

These two classes of techniques are complementary. On the one hand, inductive approaches seem to be more realistic, in particular when fault or error modeling tools used by deductive methods may leave out actual faults or errors. On the other hand, inductive approaches are often not tractable, as a huge number of functioning cases may lead the system into one given internal state. Such a problem occurs, for instance, when a program uses 'while' loop statements, as the actual number of iterations is often unknown. Inferences are then constrained by hypotheses always debatable.

Two basic concepts have been associated with the introduced approaches, namely:

- *Controllability* that defines the capability to put the system in a given functioning state, by acting on its inputs,
- *Observability* that defines the capability to observe the internal state of the system from the outputs, mainly by acting on its inputs.

These notions apply to faults, errors and failures. For instance, they estimate the capability to activate a fault, changing it into an error, and then to propagate this error to the output, as a failure.

Controllability and observability are two characteristics, which are sometimes desired, and sometimes unwanted. Thus, controllability and

observability are desired for testing, as they increase the testability of the system. On the contrary, fault tolerance required during system operation, needs to reduce fault activation, error contamination and failure occurrence, that is, to reduce global controllability and observability.

To conclude this section, let us note that the techniques dealing with dependability measurement and impairment analysis are useful at first when system design is completed. They provide means to assess the reliance that can be placed on this system. These means are also efficient during the design steps as they provide predictive tools. Design choices are validated or rejected taking the obtained measures into account. Moreover, quantitative and qualitative analyses are often handled jointly. For instance, the FMECA (Failure Modes, Effects and Criticality Analysis) is a deductive qualitative method using quantitative data. We have also noticed that the Fault Tree Method could provide quantitative results.

19.4 CHOICE OF METHODS

Among the so numerous and varied dependability methods and techniques how to choose one or several ones which are well adapted to a given target application? *Figure 19.2* caricatures some of these techniques offered to the designer. How to make an efficient tradeoff between the various and often contradictory dependability and performance criteria?

During the expression of the requirements of a product, according to the class of application considered, we can prioritize the attributes, for instance, ensuring the continuity of service (e.g. reliability) or, the safety. Let us consider as an example a robot aimed at human interactions (it could be a domestic or an industrial robot application). One can easily identify several complementary dependability requirements:

- concerning the *safety* refined into two categories:
 - safety relatively to the human (in order to prevent any injury of an operator or the public),
 - safety relatively to the process (for example, in case of manipulation of fragile objects),
- concerning the *availability* and the *maintainability* (if the product is supposed to be repairable).

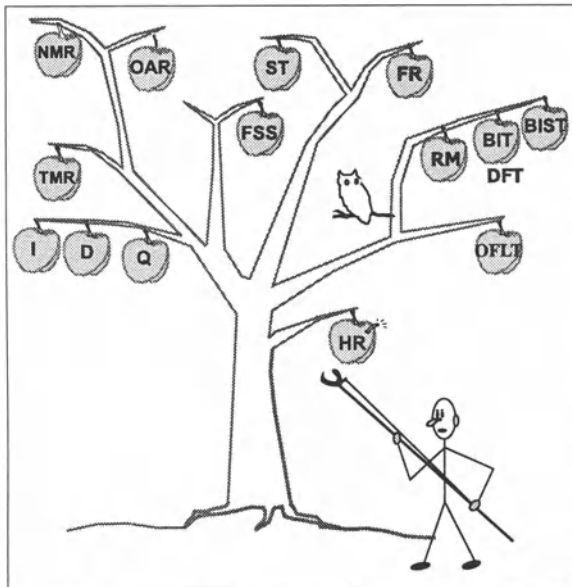
In order to satisfy these different elements of the specifications, different solutions will possibly be implemented and mixed. Thus, the safety problems can lead to

- passive solutions (choice of non aggressive technology: the ‘rubber’

robot),

- and/or active solutions (use of sensors to detect dangerous situations and implementation of emergency reaction mechanisms).

The continuity of service can be increased by the use of high reliability electronic and mechanical components and design choices facilitating the maintenance. Naturally, independently from these actions, it is obvious that the creation steps of this product must justify the required reliance placed in the final delivered service.



HF	High Reliability	FSS	Fail-Safe Systems
OFLT	Off-Line Testing	TMR	Triple Modular Redundancy (passive)
DFT	Design For Testability	NMR	N-Modular Redundancy (active)
RM	Reed-Muller circuits	OAR	Other Active Redundancy
BIT	Built-In Test	I	Interwoven Logic
BIST	Built-In Self Test	D	Dotted Logic
FR	Functional Redundancy	Q	Quadded Logic
ST	Self-Testing		

Figure 19.2. Some fruits of dependability

Appendix A

Error Detecting and Correcting Codes

In this Appendix, we compare some redundant codes:

- the single parity code (separable),
- the m -out-of- n code (non-separable), optimal for $m = \lceil n / 2 \rceil$,
- the double rail code (separable), particular case of m -out-of- $2m$ code,
- the *Berger* code (separable), optimal for $r = \lceil \log(k+1) \rceil$,
- the modified *Hamming* code (separable) which is a basic cyclic code.

We use the following notations: N is the total number of codewords, k the number of bits of the words to be coded, n the number of bits of the codewords, and r the number of redundant bits ($n = k + r$). All these parameters do not apply when the code is non-separable.

Table A.1 gives the general features of these codes and their error model (S and NS means Separable and Non-Separable).

code	parity	m-out-of-n	Double Rail (m / 2m)	Berger	Hamming modified
k, n, m, r	$k = n - 1$	$m = \lceil n / 2 \rceil$	$k = n / 2 = m$	$r = \lceil \log(k+1) \rceil$	$n = 2^{(r-1)}$
Separable	S	NS	S	S	S
N	2^k	$\binom{n}{m}$	2^k	2^k	2^k
errors detected	odd	unidirectional	unidirectional multi. / 1 rail	unidirectional	double detected single corrected

Table A.1. Basic properties

Table A.2 shows the evolution of the number of codewords that can be made when n increases. Cells noted '-' correspond to situations without interest or impossible.

n	parity	m-out-of-n	Double Rail (m / 2m)	Berger	Hamming modified
4	N = 8	6	4	4	-
5	16	10	-	8	-
7	64	35	-	16	-
8	128	70	16	32	16
9	256	126	-	64	-
10	512	252	32	128	-
11	1024	462	-	-	-
12	2048	924	64	256	-
16	32768	1287	256	4096	2048
19	262144	92378	-	32768	-
21	1048576	352716	-	65536	-
32	2147483648	601080390	65536	134217728	268435456
36	34359738368	9075135488	262144	214783648	-

Table A.2. Evolution with n of the number of codewords N

Appendix B

Reliability Block Diagrams

The Reliability Block Diagram is a very simple model used to represent redundant structures and to analyze their reliability. It was one of the first tools to be employed, and it remains pedagogically very interesting. We have introduced the principles of 'series' and 'parallel' redundant structures in Chapter 7. The aim of this appendix is to bring some complements on the reliability analysis of non-repairable redundant structures, with simple hypotheses. We assume that the modules have exponential reliability laws with constant failure rate. The reliability of the reference module is:

$$R_0 = e^{-\lambda t}, \text{ which gives } \text{MTTF}_0 = 1/\lambda.$$

1. ON-LINE REDUNDANCY

All modules are operating in parallel. As far as k of them are faultless, the system functions correctly. The value of k depends on the technique used. This corresponds to a passive redundancy according to the observability of the errors affecting each module.

Structure n modules in parallel

The system does not fail as far as one module is faultless. In the general case of n modules, we have: $(1 - R) = \prod_i (1 - R_i)$, $\text{MTTF} = \text{MTTF}_0 \cdot \sum_i (1 / i)$, where R is the global reliability, R_i the reliability of module i .

Hence, for $n = 2$ (Figure B.1), $R_i = R_0$; this gives:

$$R = 2 R_0 - R_0^2 = 2 e^{-\lambda t} - e^{-2\lambda t}, \text{ MTTF} = 1,5 \text{ MTTF}_0.$$

Numerical value: if $\lambda = 10^{-4}$, $t = 10^3$, then, $R(10^3) = 0.9909$.

For $t = 10^3$, the reliability of the reference module is $R_0(10^3) = 0.9048$; hence, $R(10^3) > R_0(10^3)$.

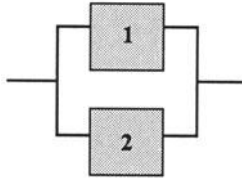


Figure B.1. Structure 2 modules in parallel

Structure *m-out-of-n*

The system does not fail as far as m modules are faultless. The outputs are elaborated by a *m-out-of-n* voter. Hence, the global reliability is:

$$R = \left(\sum_{i=m}^n \binom{n}{i} R_0^i (1 - R_0)^{n-i} \right) R_V$$

where R_V is the reliability of the voter.

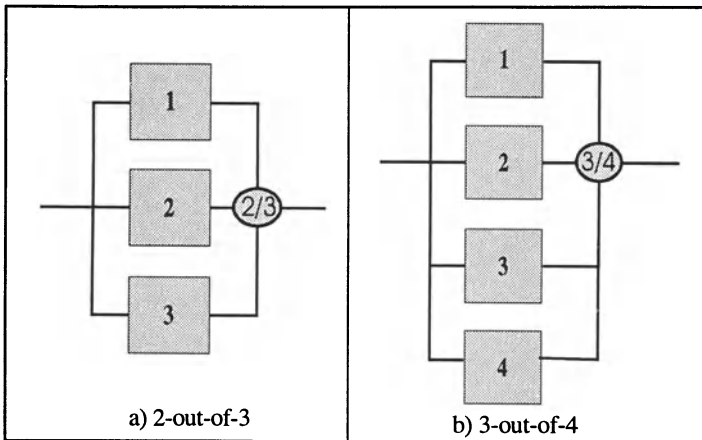


Figure B.2. Two examples of structures *m-out-of-n*

a) Structure 2-out-of-3 (called TMR)

This technique, illustrated by Figure B.2-a), has been presented in Chapter 18 as the *triplex* or *TMR*. The voter elaborates the final outputs from the outputs of the three modules. This module is supposed hereafter to be faultless. Hence, the reliability can be simply determined by enumerating the disjointed cases of good functioning:

- one case where all modules are faultless (probability: R^3),
- three cases where two modules are faultless and one module is failing (probability: $R^2(1 - R)$).

With a perfect voter, $R = R_0^3 + 3 R_0^2 (1 - R_0) = 3 R_0^2 - 2 R_0^3 = 3 e^{-2\lambda t} - 2 e^{-3\lambda t}$, $MTTF = (5/6) \cdot MTTF_0$.

The MTTF of the TMR system is lower than the MTTF of the basic module. However, these two reliability curves have an intersection point (see *Figure B.5*): for missions of small duration, the TMR has a better reliability, and for greater mission duration, the basic module is better.

b) Structure 3-out-of-4

This structure is represented by *Figure B.2-b*. For a perfect voter, we have: $R = 4 R_0^3 - 3 R_0^4 = 4e^{-3\lambda t} - 3e^{-4\lambda t}$.

Structure Double Duplex

Four modules are associated as two pairs. One of these pairs is connected to the process, while the second one is in standby. As soon as a failure is detected on the active pair, the second one replaces the defective pair. The detection results from the comparison between the two outputs of a pair. The reliability of this quadri-redundant structure corresponds to the survival probability of one of the two pairs:

$$P((1.1 \text{ AND } 1.2) \text{ OR } (2.1 \text{ AND } 2.2)) = P(1.1 \text{ AND } 1.2) + P(2.1 \text{ AND } 2.2) - P(1.1 \text{ AND } 1.2) \cdot P(2.1 \text{ AND } 2.2) = P(1.1) \cdot P(1.2) + P(2.1) \cdot P(2.2) - P(1.1) \cdot P(1.2) \cdot P(2.1) \cdot P(2.2)$$

(all probabilities are independent).

Hence, as all modules have the same reliability: $R = 2 R_0^2 - R_0^4 = 2 e^{-2\lambda t} - e^{-4\lambda t}$.

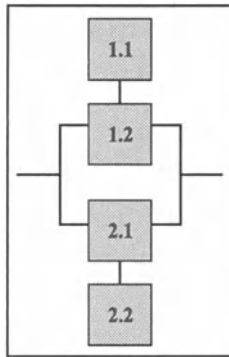


Figure B.3. Double Duplex

2. OFF-LINE REDUNDANCY

An *off-line* redundant structure possesses n elements; one of them is connected to the process, while the $(n - 1)$ other modules are in off-line or *cold standby*. We assume that faults only occur in active modules; thus, the reliability of the redundant standby modules is supposed to be perfect. When the active module is failing, it is

replaced by a standby module. The detection and reconfiguration mechanism is not considered in the reliability block diagram: it is supposed here to be faultless.

As the events are not independent, the reliability calculus is made easier by the use of the *Laplace* transform. We obtain:

$$R = e^{-\lambda t} \cdot \sum_{i=1}^n ((\lambda t)^{i-1}) / (i - 1)!, \text{ MTTF} = n \cdot \text{MTTF}_0.$$

Special case: $n = 2$

$$R = e^{-\lambda t} + \lambda t e^{-\lambda t}, \text{ MTTF} = 2 \cdot \text{MTTF}_0,$$

Numerical value: if $\lambda = 10^{-4}$ and $t = 10^3$, then $R = 0.9953$.

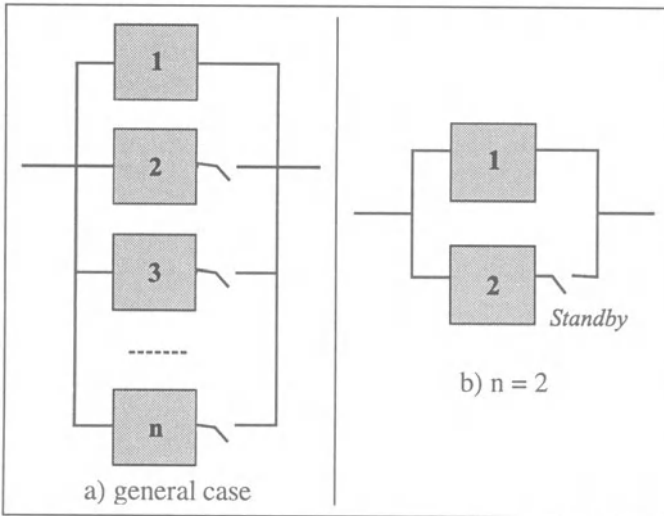


Figure B.4. Off-line redundancy

3. COMPARISON OF SOME STRUCTURES

To conclude this Appendix, we show in *Figure B.5* the reliability curves of some redundant structures:

- TMR,
- 2 modules in parallel,
- 3-out-of-4,
- Double Duplex,
- and off-line redundancy with $n = 2$.

These curves are drawn for $\lambda = 10^{-4}$, and they are referred to the reliability of a single module called 'basic module'.

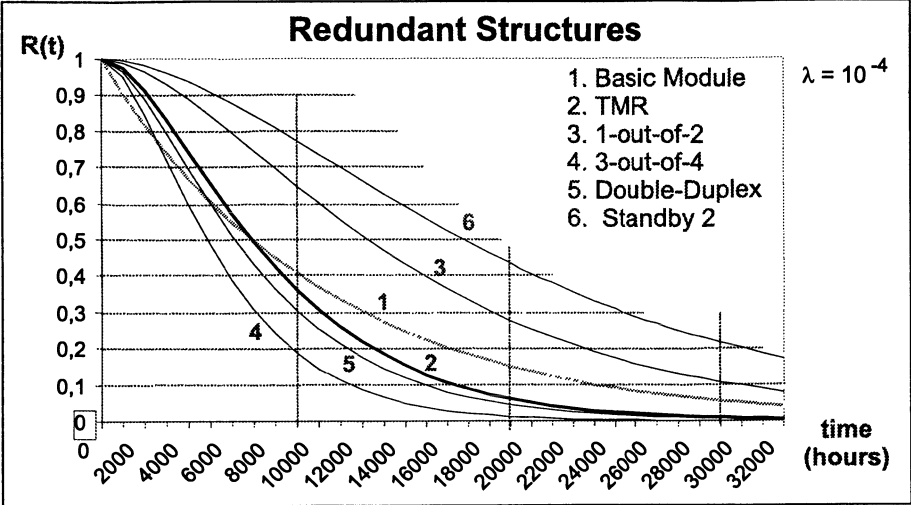


Figure B.5. Reliability of some redundant structures

Appendix C

Testing Features of a Microprocessor

All semiconductor manufacturers are very much interested in the dependability features of the components they produce (ASICs, microprocessors, micro-controllers, etc.). Naturally, this interest deals with all design and fabrication aspects of their products: use of fault prevention and fault removal techniques during specification, design and fabrication stages. In this general dependability framework, production but also maintenance *test* has a predominant place. In particular, *design for testability* techniques are integrated in the design process. We have already mentioned the present IEEE 1149-1 boundary scan standard.

Some semiconductor companies go a step ahead in that direction and consider critical applications with the use of redundant microprocessor structures. We report in this appendix some very general information about the *Pentium* microprocessor, interpreted from an *Intel* documentation. The interest of this presentation is to show the rather great variety of dependability techniques offered by a general purpose integrated circuit. Naturally, such an approach can be found in many other concurrent circuits such as those produced by Motorola, AMD, etc. We will identify three main levels of means: Aid to the debugging of microprocessor applications, Off-Line testing, and On-Line testing.

1. DEBUGGING AID

During the debugging of an application running on a microprocessor, it is necessary to understand the execution of the implemented programs. Like all processors, the *Pentium* offers a debugging mode called 'Probe Mode' which allows accessing from the outside to the internal registers, to the system memory I/O spaces, and to the internal state of the microprocessor. The *Pentium* has 4 debugging registers used to insert breakpoints. Moreover, the specialist in charge of the debugging can access to internal counters that records some events of the internal evolution. All these features obviously belong to the design verification group of techniques.

2. OFF-LINE TESTING

The *Boundary Scan* (IEEE 1149.1) standard has been implemented in the microprocessor for testing at 'global board level'. This means that in any system comprising a microprocessor connected to other circuits (such as memory unit, interface circuits, etc.) on a PCB, it is possible to access through the Pentium to these others circuits in order to apply test sequences to them and to collect the resulting outputs.

The following pins of the test bus are accessible: *TCK*, *TDI/TDO*, *TMS*, *TRST*, as well as the test logic (the *TAP* automaton).

Finally, the *Pentium* integrates a *BIST* procedure that is automatically executed when the microprocessor is switched on. This *off-line testing* procedure is called *Reset Self-Test* but in reality the term 'self-test' refers here to a *Built-In Self-Test* technique. *Intel* announces that this integrated test covers 100% of the single stuck-at 0/1 faults of the Micro-Code PLAs, memory caches (instruction and data caches), and some other internal circuitry (TLB, ROM).

3. ON-LINE TESTING

This component also offers *on-line testing* features:

- internal on-line error detection, thanks to error detecting codes,
- redundancy capability allowing Duplex redundant structures.

Error Detection

During the functioning of the Pentium, some error detection mechanisms are activated by a specialized automaton called the *Machine Check Exception*. These errors are revealed by the use of single parity error detecting codes:

- single parity test on the Data Bus (DATA PARITY):
64 bit-Data Bus + 8 parity bits (one bit per byte of data),
- single parity on the Address Bus (ADDRESS PARITY):
32 bit-Address Bus + 1 parity bit,
- some other internal parity codes.

Microprocessor Redundancy

Finally, the circuit has been designed in order to allow a Duplex redundant structure to be easily implemented, thanks to the *Functional Redundancy Checking (FRC)* technique. *Figure C.1* illustrates this technique.

A 'Master' microprocessor performs the normal functioning of the application and is connected to the external process.

A second microprocessor, called 'Check', plays the role of an *observer*. When an error is detected (by simple comparison of the two functions), the output signal IER is activated, calling for an external action (alarm, switch-off, recovery, etc.). The commercial document of Intel ensures that more than 99% of the faults are thus detected.

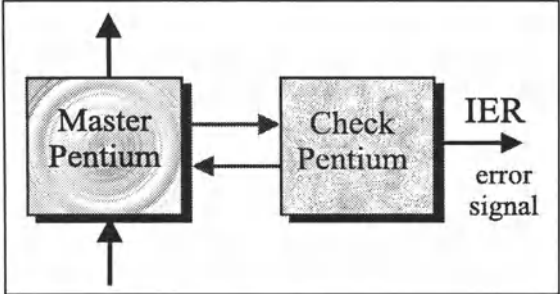


Figure C.1. Duplex Structure

Appendix D

Study of a Software Product

Ariane V Flight Control System

The first launch of *Ariane V* led to the destruction of the rocket, due to a failure of the embedded computing system. Whereas most of the firms whose projects failed had hidden the causes, the CNES (French national space agency) provided numerous pieces of information whose study concurred in the improvement of knowledge on dependability. The following presentation is based on the published documents. The analysis developed in this appendix must above all strengthen the opinion that the mastering of faults in complex computing systems is very difficult, illustrating this idea on a real example.

1. FAILURE SCENARIO

A simplified view of the architecture of the computing system embedded in the rocket is provided in *Figure D.1*.

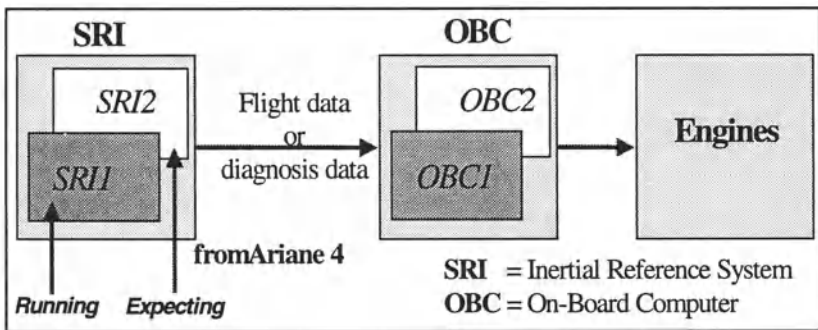


Figure D.1. Architecture of the Control System of Ariane V

The engines (Vulcan main engine and boosters) are controlled by the OBC (On-Board Computer) which receives data from various sensors which are autonomous complex sub-systems. The SRI (Inertial Reference System) is such a sub-system. It provides flight data concerning the rocket position. The OBC as well as the SRI have redundant hardware boards based on a *recovery block*. The first hardware board is in operation till an error is detected; then, the second board replaces the first one. These hardware systems execute complex software real-time applications using a multitasking kernel. The programs executed on the two hardware platforms SRI1 and SRI2 or OBC1 and OBC2, are the same.

Among its numerous treatments, the program of the SRI (SRI1 or SRI2) calls a function which make a conversion between a real value expressed in a particular format and an integer value. This function was previously used in the software managing the flight control of *Ariane IV*. Being dependent on the acceleration, the actual values handled by this function at *Ariane IV* launch time were in a given range. Unfortunately, the acceleration of *Ariane V* being higher, the conversion function was called with a value out of this range. This situation raised an exception during the function execution.

The fault-tolerance mechanism implemented in SRI1 handled this erroneous state, switching on the SRI2 redundant system. Executing the same program, the same exception raised. Its handling by SRI2 consisted in communicating a diagnosis data to the OBC before switching off the SRI system. Thanks to this information, the OBC should continue the flight in a *degraded mode*, for instance extrapolating the evolutions of the rocket positions. Unfortunately, the diagnosis data communicated by the SRI2 were interpreted by the OBC as a flight data. Thus, the OBC reacted by swiveling the engines.

2. ANALYSIS

2.1. Fault Diagnosis

The first question raised is “who is responsible?” that is, “where is the fault?” The conversion function seems to be the obvious guilty: it was unable to convert the given data. Is it so simple? We may also consider that the failure comes from:

- the OBC because it interpreted a failure identification as a flight data,
- the OBC which did not perceived these flight data as erroneous, for instance, by likelihood check,
- the SRI2 as it did not provide correct flight data,
- the SRI which did not tolerate a software fault,
- the conversion function which raised the exception,
- the too important acceleration of the rocket at launch time,
- the development team which did not detect the presence of a fault,
- the managers who required the reuse of this part of *Ariane IV*, etc.

So, it is difficult to adjudge the fault to a part of the system or to a partner of the project. However, this example illustrates several aspects highlighted in the book.

2.2. Fault Prevention

At first, the specification is an important phase of the development. The specification model must define the role of the system but also the domain in which the services will be provided. For instance, the constraints associated with the values of the input parameter of the conversion function were not precise.

Secondly, the presence of redundant elements may be dangerous if redundancy is not mastered. For example, the conversion function input presents a large functional redundancy: the integer and real types constitute the Universes whereas the Static Domains were reduced to ranges. On the contrary, the use of one output parameter of SRI for two concepts (the flight data and the diagnosis data) was possibly due to performance reasons. As several times mentioned, the dependability requirements are often against the performance requirements (time, memory, costs, etc.). So, a compromise must be found to develop industrial dependable real-time systems.

The fact that the conversion function is a component successfully used in *Ariane IV* shows the difficulties of *reuse*. A priori, the reuse of a component increases the reliance which can justifiably be placed on the service it delivers, that is, its dependability. However, the justification of this reliance was obtained for a specific functional and non-functional environment; this reliance was not preserved when the environment changed.

The *Ariane V* control system is a complex system in which numerous elements interact: hardware platforms interact with software applications to detect errors and to handle them (switching from the initial platform to the redundant one), complex coupling between the sub-systems SRI and OBC, etc. These complex interactions are at the origin of numerous faults in the recent systems which use the integration of sub-systems. Each sub-system operates correctly, whereas errors occur when the sub-systems interact each other. Most of the errors propagated during the first flight of *Ariane V* are coming from integration issues: SRI2 interprets the exception raised by SRI1 as a hardware failure and OBC interprets the diagnosis data as a flight data.

2.3. Fault removal

The reader is probably amazed that the fault of the conversion function was not detected during the reviews and test procedures. Probably, as previously mentioned, the reuse of the SRI was the cause of less checking. The successful use of a component during several years increases an unjustified reliance on the component, and then decreases the time and the money spent for its testing.

To be efficient, the fault detection in a component requires the handling of information on the component domain. In particular, the coverage of 100% of a structural testing may not detect any faults due to use out of the domain. The structural test should take into account the domains of the components used to implement the system: can a sequence lead the system operation to reach states out of the domain? Consider, for instance, the following statement:

```
K := . . . I * J . . . ;
```

The assessment of the program must detect values of I and J such as I*J provokes an overflow, that is, the result of this multiplication is greater than the higher integer which can be expressed by the run-time resources.

Finally, the complexity of the global system and of the physical devices of its functional environment often limits the integration testing.

2.4. Fault Tolerance

After the failure, most of the criticisms concerned the exception mechanism which raises the error. Several persons proposed the suppressing of the raising to continue the execution. This viewpoint is dangerous. The absence of error detection does not prevent the occurrences of errors; it just allows masking them. In this case, as well as when the exception handler consists in doing nothing, the behavior of the system is hazardous and causes an uncontrollable contamination of the errors in the system. This situation is illustrated by the relationships between the SRI and the OBC: the SRI signals its failure considered as a normal data. The use of an exception would force the OBC to take this information into account.

The wrong but not detected communication between SRI and OBC also shows the importance of redundant elements to *detect error on-line*. For instance, the use of likelihood checking certainly would have detected the flight data inconsistency. In the same way, redundant information would be useful to detect that the exception raised in SRI1 was not due to a hardware failure but software one. This diagnosis probably leads to another reaction to handle the error.

The described scenario presents a contamination of errors: from the conversion function to SRI1 then to SRI, OBC1 and the engines. To handle the problems coming from integration of components, the designer must pay a special attention to the *error confinement*.

Finally, the fact that only hardware faults were tolerated illustrates that engineers or managers assume that the problems come from the aggressions of the environment. Of course, these causes exist, particularly for spatial applications. However, the *Ariane V* failure shows that the human is also often at the origin of the faults.

Appendix E

Answer to the Exercises

FIRST PART

Exercise 3.1. Failures of a drinks dispenser

1. Static failure: the money change or return operations are incorrect.
2. Dynamic failure: when the machine has delivered a coffee, the red light stays on one minute before authorizing the next drink to be selected.
3. Temporary failure: this morning, the machine was unable to deliver tea.
4. Static and persistent failure: the $\frac{1}{2}$ \$ coins are no longer accepted by the machine.

Exercise 3.2. Faults of a drinks distributor

1. Examples of functional and hardware faults.

Functional fault. The machine does not test the state of one of the resources (coffee, tea, chocolate, cup, sugar, and spoon). Consequently, the service is no longer delivered to the user who has paid and selected his/her drink. He/she obtains an empty cup. This is a *persistent* and *static* failure. Thus, the ‘money manager’ automaton which uses the state diagram is correctly functioning. On the contrary, the ‘drink delivery’ automaton which interprets the orders and delivers the drink does not execute the order.

Hardware fault. A fault of the tea selection button may lead to a quite different failure. If this button become inactive, the machine does not receive any tea selection order, so it stays in the ‘selection’ state, waiting for a user selection. This user who would like to drink tea has to cancel or select another drink (for example a coffee). The finite state machine cannot directly pass from the global ‘selection’ state to the ‘delivery’ state allowing for a tea selection.

This fault is equivalent to the cut of the arc connecting the state ‘selection’ to the state ‘delivery’ for a tea selection.

2. Money management alteration. Money is implied in states 'payment', 'cancel' and 'change'. Hence, functional and hardware faults altering this money service are to be found in these states. The panel of possible faults is rather large: coin rejection, impossibility to cancel a drink, incorrect money change, etc.
3. Functional transformation. The new proposed functionality can be obtained by modifying the global functional state graph as shown in *Figure E.1*. When the drink is delivered, its price is subtracted from the value of the total amount of money introduced in the machine; then, the system comes back in the 'selection' state. The user can then choose a new drink or order the return of the remaining money (by pressing the 'cancel' button).

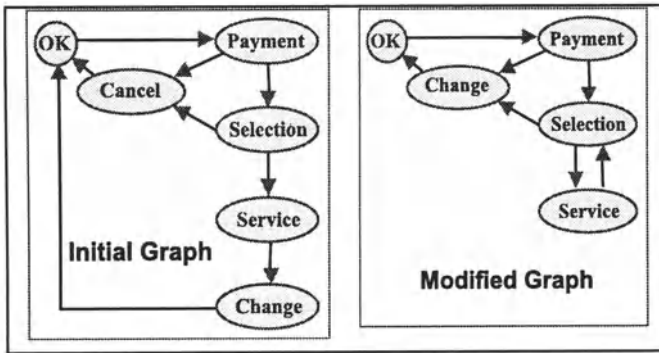


Figure E.1. Initial graph and modified graph

Exercise 3.3. Study of a stack

1. In order to simplify the study, we suppose that the real size of the stack is only 10 objects. We will now study some faults.

Design fault. The real size of the stack has been underestimated (for instance, 10 locations only), whereas the storing of 15 objects is envisaged. If 15 objects are pushed, the fault produces a failure. Assuming that the stack memorizes integers, let us consider the program:

```

for i varying from 1 to 15, loop
    PUSH A(i);
end loop;

```

If the `Stack_Full` mechanism is used, the preceding treatment will stop at the first stack overflow. This overflow can raise an exception in the case of a software implementation. Hence, the fault provokes a failure which is detected.

Hardware fault. If a breakdown affects the `Stack_Full` signal and maintains it at '0' (no signaling) despite excessive stacking, the calling program can send too many different values to store. In the case of the preceding program, what will occur after the 10th value sent to the stack?

If the stack refuses to store more than 10 values, 5 values will remain not stored. The stack could accept all coming values and store the 5 values 11 to 15 at the 10th memory address. Or else, the stack could return to the first address and store the values 11 to 15 at the addresses 1 to 5, hence erasing the preceding stored data.

These possibilities depend on the implementation of this stack: hardware with gates and registers, or simulation of the stack in the main memory.

A test sequence detecting this fault could be to Push 15 integers (from 1 to 15), and then to Pop 15 values and to compare them with the initial values. Let us note that this test sequence also detects the previous functional design fault.

External fault. The user of the stack ignores the signal indicating an overflow. The sequence given for the case of design fault will transform this fault into a failure. In both cases, the failure is the same, (but without any detection).

2. A normal use of a stack is to apply a same number of Push operations than Pop operations. If a fault provokes the application of a Pop action to an empty stack, a failure occurs. The situation is very similar to the overflow studied in the previous question, and the use of `Stack_Empty` signal allows the detection of such situation.

Exercise 3.4. Study of a program

For an addition such as $Exp1 + Exp2$, where $Exp1$ and $Exp2$ are two arithmetic expressions, the compiler generates a sequence of executable instructions allowing the evaluation of these expressions. The two obtained results are placed in two distinct registers. Then, the compiler adds an instruction which perform the sum of the content of these registers. However, the programming language does not define the order of evaluation of these two expressions: one can first evaluate $Exp1$, then $Exp2$, or the opposite! This means that, for our example, we will compute first $F1$, then $F2$, or the opposite.

Let us examine the functioning with '1' as the initial value of A .

After execution of $F1$, $A = 2$, which is also the value returned by $F1$. Then, after execution of $F2$, the value of A and the value returned by $F2$ are equal to 4. Hence, B will then take the value: $2 + 4 = 6$. On the contrary, if $F2$ is evaluated first, the value of A and the value returned by $F2$ are equal to 2 (A being initially equal to 1). Then, the execution of $F1$ returns 3. Hence B will be equal to $2 + 3 = 5$.

Consequently, according to the executable code generated by the compiler, the final result of B is either 6 or 5!

What could be concluded from this analysis? The addition is a commutative operation, so both interpretations of the compiler are acceptable. However, this commutativity property is only effective for the addition of values (i.e. $5 + 3 = 3 + 5$), and not for the addition of expressions having 'side effects' (in our example, the execution of $F1$ and $F2$ modify A). Thus, a possible failure (only one of these two interpretations is expected) may result from the fact that the designer does not know how the technology he/she uses will operate on the source code. Here, the technology deals with the implementation of the program by the compiler.

Exercise 4.1. Latency of an asynchronous counter

The MSB (Most Significant Bit) will normally switch to the value '1' after 8 clock pulses. Hence, the latency is equal to $8 \times 2 \text{ ms} = 16 \text{ ms}$.

The fault will lead to a failure that remains 8 clock pulses and then disappear.

Exercise 4.2. Latency of a structured system

Error #1: 10ms, error #2: 110ms, error #3 = failure of the system: 140ms.

Exercise 4.3. Consequences of failures

Mean cost = $(2 \times 0 + 3 \times 5 \cdot 10^3 + 2 \times 6 \cdot 10^3 \times 4 + 3 (10^3 + 3 \cdot 10^3 \times 4)) / 10$.
 Mean cost = 10.2 ku.

Exercise 4.4. Fault - Error - Failure in a program

1. As the actual last right page number is 325, the expected result is $(325+1)/2 = 163$ sheets.

If the faulty expression provides 326 instead of 325, the result computed is $(326+1)/2 = 163$, taking the integer division semantics into account. So, the result is good: no failures occur.

Considering 327, the returned value is $(327+1)/2 = 164$. The procedure execution fails. The same result and conclusion is obtained with 328.

2. An error characterizes an unacceptable state or state evolution occurring at run-time. The states being characterized by values taken by attributes, consider Last-Right-Page as attribute. "The last right page number of a book is odd" is a property. Therefore, no error is detected in the first case (326), no error is detected in the second case (327), and an error is detected in the third case (328).
3. **Conclusion.** Consider *Table E.1* which synthesizes the three cases. A fault may provoke an error which may provoke a failure. In the first case, the fault is activated as an error, but the last statement tolerates it (no failure occurs). In the second case, no error is detected, as the observation means is not sufficient. The property which characterizes the error is not accurate enough: all even values are not correct values. However, the program fails. The last case is the conventional one: the fault activates an error, which propagates as a failure.

case	fault	error detection	failure
1	yes	yes	no
2	yes	no	yes
3	yes	yes	yes

Table E.1. Fault - Error - Failures cases

Exercise 5.1. Faults of a MOS network

The determination of the logical expression of a 'structured' MOS network can be obtained by an iterative decomposition of this network into 'series' and 'parallel' sub-networks, till reaching the basic MOS components.

1. We perform this analysis for the fault-free network and for the two faulty networks:
 - Faultless circuit: $R = a b' + b c$,
 - Circuit with fault $F1$: $R_1 = b c$,
 - Circuit with fault $F2$: $R_2 = (a + c) (b + b') = (a + c)$.

The 3 corresponding functions, N (faultless circuit), N_1 (with fault $F1$) and N_2 (with fault $F2$), are shown in *Table E.2*. The output takes the same values for four input configurations only: 000, 010, 011, and 111.

a b c	N	N ₁	N ₂	N ₃	N ₄
0 0 0	0	0	0	0	1
0 0 1	0	0	1	0	1
0 1 0	0	0	0	0	0
0 1 1	1	1	1	1	1
1 0 0	1	0	1	1	1
1 0 1	1	0	1	1	1
1 1 0	0	0	1	0	0
1 1 1	1	1	1	1	1

Table E.2. Normal and erroneous functions

2. N_2 becomes $N_3 = (a + b) \cdot (b' + c) = a b' + a c + b c = N$, without fault. Thus, this fault has no influence on the function performed by the network (see Table E.2).

Let us note that in the faultless circuit, the permutation between transistors controlled by b and c has no influence on the functioning. However, the same fault F_2 will have different effects, (failures) according to the chosen network!

3. Fault F_3 : the function becomes $N_4 = b' + b c = b' + c$, shown on the previous table. It induces two failures highlighted in bold.

Exercise 5.2. Faults of a full adder

1. **Functional fault F_1 .** The ‘sum’ output (S) is not modified, but the ‘carry’ (C) becomes $C1 = a b + a' b' c$. There are three failures on the carry output, for $abc = 001, 101$ and 011 .
2. **Hardware fault F_2 .** The classical *Stuck-At 0/1* fault model supposes that a fault occurring on a gate input line has no backwards effects. Here, the NAND gate receiving a and b is not altered by the fault α .

The input b no longer acts on the output S , producing 4 failures. The carry function becomes $C_2 = [(ab)' \cdot ((a \oplus 0).c)']' = a b + a c$, instead of $ab + ac + bc$. A failure occurs for $a' b c = 1$.

a b c	C S	C ₁ S ₁	C ₂ S ₂
0 0 0	0 0	0 0	0 0
0 0 1	0 1	1 1	0 1
0 1 0	0 1	0 1	0 0
0 1 1	1 0	0 0	0 1
1 0 0	0 1	0 1	0 1
1 0 1	1 0	0 0	1 0
1 1 0	1 0	1 0	1 1
1 1 1	1 1	1 1	1 0

Table E.3. Truth tables: without fault and with faults F_1 and F_2

Table E.3 gives the output values without fault and with faults F_1 and F_2 . The values noted in bald characters show the failures.

3. The two faults produce quite different failures. It is possible to distinguish between these faults by applying to the circuit an input vector such as 011. The diagnosis is as follows:
 - if the output C only is erroneous, then fault F_1 is present,
 - if outputs C and S are erroneous, then fault F_2 is present,
 - if both outputs are correct, none of these two faults is present.

Exercise 5.3. Fault models and failures

1. We draw the truth table associated with each fault. Thus, the erroneous function for c stuck at 0 (noted c^0) is $z = a b$; it provokes 3 failures for the input vectors 001, 011, and 101. We observe on Table E.4 that faults a^0 , b^0 and d^0 are equivalent, and that faults d^1 , c^1 and z^1 are equivalent.

a b c	z	a ⁰	a ¹	b ⁰	b ¹	c ⁰	c ¹	d ⁰	d ¹	z ⁰	z ¹	FF1	FF2
0 0 0	0	0	0	0	0	0	1	0	1	0	1	1	1
0 0 1	1	1	1	1	1	0	1	1	1	0	1	1	0
0 1 0	0	0	1	0	0	0	1	0	1	0	1	1	1
0 1 1	1	1	1	1	1	0	1	1	1	0	1	1	0
1 0 0	0	0	0	0	1	0	1	0	1	0	1	1	1
1 0 1	1	1	1	1	1	0	1	1	1	0	1	1	0
1 1 0	1	0	1	0	1	1	1	0	1	0	1	0	0
1 1 1	1	1	1	1	1	1	1	1	1	0	1	1	0

Table E.4. Correct and erroneous functions

2. Let us assume that the functional faults can affect each gate by transforming it into any other gate type: AND, OR, NOT, NAND and NOR. We illustrate these faults with two cases (Table E.4):

- FF1 which transforms the AND gate into a NAND gate:
 $z = (a b)' + c = a' + b' + c.$

- FF2 which transforms the OR gate into a NOR gate:
 $z = (a b + c)' = a' c' + b' c'.$

These two failures do not belong to those induced by the stuck-at fault model. Going further, we can wonder if this functional gate-transforming model is able to complement the set of all theoretical failures (255 classes!). The answer is not: for instance, the erroneous function $z = a b' + b c'$ cannot be obtained with these fault models. Now the question not answered here is:

Can such a failure occur, and from which technological or functional faults?

Exercise 5.4. Faults of a sequential circuit

From the circuit, we can write the logical expressions of the D-inputs of the Flip-Flops; then, we deduce the transition table and the state graph shown in *Figure E.2*.

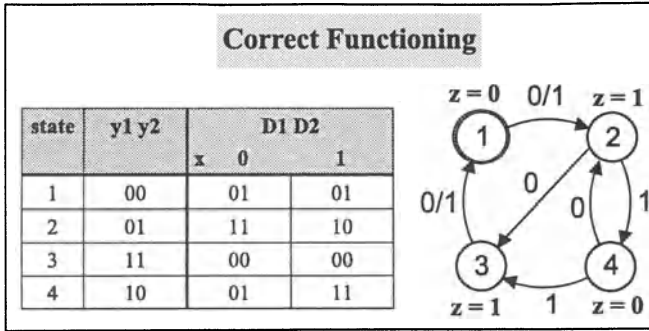


Figure E.2. Transition table and state graph of the correct circuit

1. The transformation of gate *A* into a NOR modifies the logical expression of *D2* which becomes $D2 = y2' + x' + y1'$. *Figure E.3* shows the new transition table and state graph. We observe that two transitions are modified: the arc joining state 2 to state 4 when $x = 1$ is now going to state 3, the arc joining state 3 to state 1 when $x = 0$ is now going to state 2.

This analysis led us to represent the initial functional fault at ‘state graph level’ by a new fault model (arc modification). If we suppose that state 1 is the initial state, then by applying the input sequence $\langle 0, 1 \rangle$, the circuit goes into state 2 and finally state 3 instead of state 4; the final output is $z = 1$ instead of $z = 0$: hence a failure occurs.

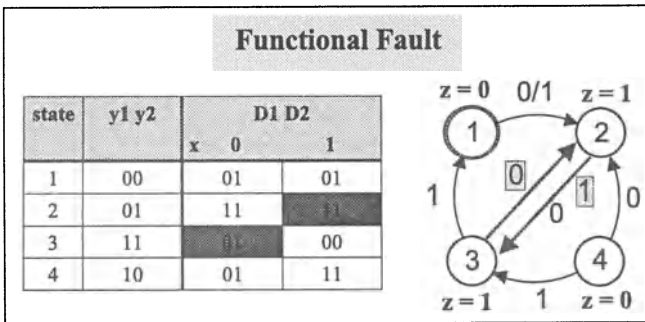


Figure E.3. Influence of the functional fault

2. The ‘stuck-at 1’ fault noted α modifies the logical expression of *D1* which becomes: $D1 = y1.y2' + y1'.y2$. *Figure E.4* shows the new transition table and state graph. Only one transition is modified: the arc joining state 4 to state 2 when $x = 0$ is now going to state 3. Here also, we have transformed the hardware fault model into a graph fault model.

If we apply the input sequence $\langle 0, 1, 0 \rangle$ to the initial state 1, the system goes into states 2, 4 and 3 instead of state 2. However, no failure occurs at the output z ! A failure is produced if a new vector $x = 0$ is added to this sequence: the incorrect circuit reaches state 1 instead of state 3 and gives a final output $z = 0$ instead of 1.

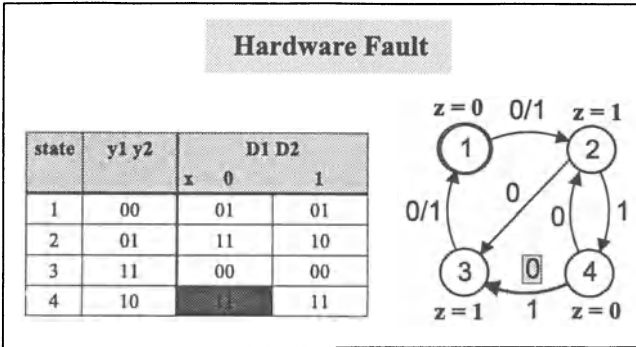


Figure E.4. Influence of the hardware fault

Exercise 5.5. Software functional faults

1. Fault analysis: **Line 5:** $Sum := A(i) - Sum;$

The Sum is iteratively subtracted from each value $A(i)$. Then, we divide the result by the number of values. We obtain a final Sum value = -15 and a final Average value = - 3.75 instead of + 3.25. The difference between the correct value and the erroneous one is quite important; hence, the external consequences of such a failure can be serious. However, this difference depends on the values stored in the array A. For example, if we add a fifth figure equal to 0, the erroneous average becomes 3 instead of 2.6: thus the difference is only 0.4 in that case!

Line 7: $return Sum / (A'last - A'first);$

This fault provokes a bad counting of the total number of numbers to be averaged: correct number minus 1. The seriousness of the resulting failure decreases with the number of values to be considered. Consequently, this fault has more 'regular' effects than the preceding one.

2. The result becomes - 7.375. The performed mathematical function is transformed:

$$T = 2^{-1} A(4) - 2^{-2} A(3) + 2^{-3} A(2) - 2^{-4} A(1).$$

Exercise 5.6. Software technological faults

The proposed program converges when N increases, whereas the series mathematically diverges. This failure comes from the limited precision used to represent the floating numbers in computers. When I reaches a certain value, $1.0 / float(I)$ is computed as 0.0. This situation is an example of technological fault, as the real number representation differs from their mathematical definition.

Let us note that for most programs, this fault actually exists but has no serious effects, even if division operations are used. Such a situation occurs in exercise 5.5, as the value of N does exceed the precision limit.

SECOND PART

Exercise 7.1. The ‘fault – error – failure – detection – repair’ cycle

1. Figure E.5 shows the interpreted cycle:

- Le_1 : latency of the fault according to the occurrence of the first error,
- L_f : latency of the fault according to the occurrence of the failure,
- D : detection time, R : repairing time,
- SF : mean time of good functioning to the occurrence of a fault.

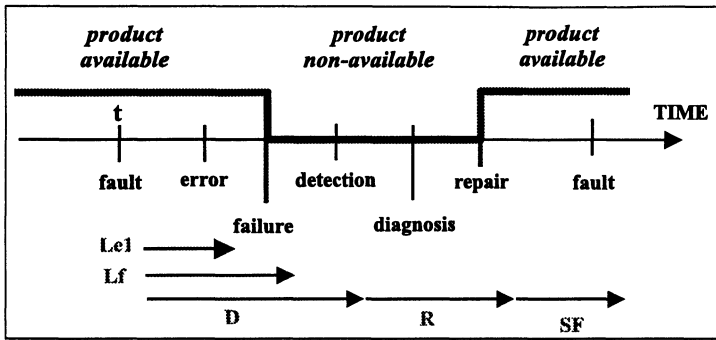


Figure E.5. Cycle of a repairable product

2. **MTBF study.** $MTBF = L_f + SF$: mean time to the occurrence of a failure.

The integration of the latency phenomenon increases the MTBF/MTTF:

$$MTTR = (D - L_f) + R.$$

The *availability rate* of this system is: $(SF + L_f) / (D + R + SF)$.

Exercise 7.2. Reliability of a component

The component follows an exponential reliability law with a constant failure rate λ .

1. We perform a definite mathematical integration of $R(t) = e^{-\lambda t}$, from 0 to ∞ . The mean time is $1 / \lambda$, that is to say 10^6 hours.

The reliability at the mean time is $R(1 / \lambda) = e^{-1}$.

2. $dR / dt = - \lambda R$. It is the derivative of the survival law, that is to say the failure density at time t . The tangent at the origin is given by the equation: $y = \lambda x + 1$; this line met the abscissa at time $1/\lambda$.

3. $\lambda = - (dR / R) / dt$. It corresponds to the conditional probability of a fault occurring at time t during a time unit (1 hour).

4. The second version is more reliable than the first one, as it has a smaller failure rate: $R_2(10^4) / R_1(10^4) = e^{0,099} = 1,1041$.

Note. The failure rate has been multiplied by 10, but the reliability at time 10^4 H is multiplied by 1.1 only.

Exercise 7.3. Composed reliability

1. **Series diagram.** The global reliability function is the product of the reliability functions of the constituting modules:

$$R(t) = R1(t) \cdot R2(t) = e^{-\lambda_1 t} \cdot e^{-\lambda_2 t} = e^{-(\lambda_1 + \lambda_2) t}$$

Hence, the failure rates of the components are added:

$$\lambda = \lambda_1 + \lambda_2, \text{ MTBF} = 1 / (\lambda_1 + \lambda_2).$$

If $\lambda_1 = \lambda_2$, the MTBF is divided by 2. We note that the value of the MTBF is inversely proportional to the number of components (if they are identical).

2. **Parallel diagram.** $1 - R(t) = (1 - R1(t)) \cdot (1 - R2(t)).$

$$\text{Thus, } R(t) = R1(t) + R2(t) - R1(t) \cdot R2(t).$$

$$\text{MTBF (or MTTF)} = 1 / \lambda_1 + 1 / \lambda_2 - 1 / (\lambda_1 + \lambda_2).$$

If the two components are identical, $R(t) = 2R_0 - R_0^2$ (where R_0 is the reliability of each one). $\text{MTBF} = 1,5 \text{ MTBF}_0.$

3. Reliability of one module: $R_0 (10^3) = 0,9048.$

Series diagram: $R(10^3) = e^{-0,2} = 0,8187 \rightarrow$ the reliability is smaller.

Parallel diagram: $R(10^3) = 0,9909 \rightarrow$ the reliability is higher.

Exercise 7.4. Comparison of two redundant structures

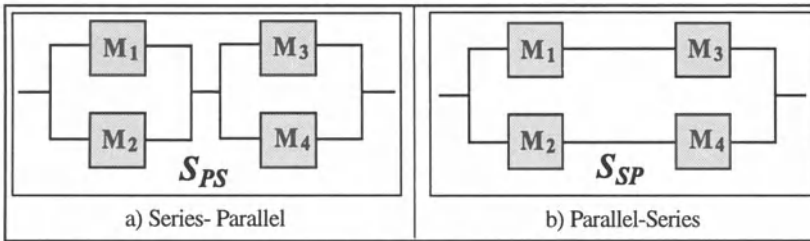


Figure E.6. Two redundant structures

1. **Structure 'parallel-series':**

$$R_{ps} = (1 - (1 - R1) (1 - R2)) (1 - (1 - R3) (1 - R4)),$$

$$R_{ps} = R^4 - 4 R^3 + 4 R^2, \text{ if } Ri = R.$$

Structure 'series-parallel':

$$1 - R_{sp} = (1 - R1 \cdot R3) (1 - R2 \cdot R4),$$

$$R_{sp} = 2 R^2 - R^4, \text{ if } Ri = R.$$

2. Comparison of the two structures:

$$R_{ps} - R_{sp} = 2 (R^2 - R)^2 \text{ which is always positive; so } R_{ps} > R_{sp}.$$

Thus, the first structure is always more reliable than the second structure.

Note. As the faults altering the modules are independent, these reliability results can also easily be determined by the composition reliability theorems. For example, for the PS structure we have: $R_{ps} = P((1 \text{ or } 3) \text{ and } (2 \text{ or } 4)) = P((1 \text{ or } 3) \cdot P(2 \text{ or } 4)) = (P(1) + P(3) - P(1) \cdot P(3)) \cdot (P(2) + P(4) - P(2) \cdot P(4)) = (2R - R^2)^2$, if all modules have the same reliability R .

Exercise 7.5. Safety analysis by a Markov graph

The evolution matrix which gives the probability to pass from a state to another (with a sampling rate expressed by hour) is shown in *Figure E.7*. After two elementary periods (hours), the probability to reach state 4 (considered as dangerous) is equal to $p1.p3 + p2.p4$. The raising of this matrix to the successive power of 2, 3, etc., gives the progression of the probability values to reach this dangerous state (hour after hour). As this system does not possess any regeneration mechanism, all parameter values always increase and are bounded by 1; this means that the degradation probabilities increase with time.

$$P = \begin{bmatrix} 1 \rightarrow 1 & 1 \rightarrow 2 & 1 \rightarrow 3 & 1 \rightarrow 4 \\ 2 \rightarrow 1 & 2 \rightarrow 2 & 2 \rightarrow 3 & 2 \rightarrow 4 \\ 3 \rightarrow 1 & 3 \rightarrow 2 & 3 \rightarrow 3 & 3 \rightarrow 4 \\ 4 \rightarrow 1 & 4 \rightarrow 2 & 4 \rightarrow 3 & 4 \rightarrow 4 \end{bmatrix} = \begin{bmatrix} (1-p1-p2) & p2 & p1 & 0 \\ r2 & (1-p4-r2) & 0 & p4 \\ r1 & 0 & (1-p3-r1) & p3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure E.7. Evolution matrix

Exercise 7.6. Representation of a system by a stochastic Petri net

Figure E.8 shows the failing and restoring mechanisms of this system. When an active unit fails and if the spare is available, the spare unit replaces the failing unit with a rate ρ . This failing unit is then symbolized by a token in place $P5$, waiting for repairing (with rate μ). Then, it is considered as the new spare unit (a token in place $P3$). The spare unit is submitted to failures and repair with rates λs and μs .

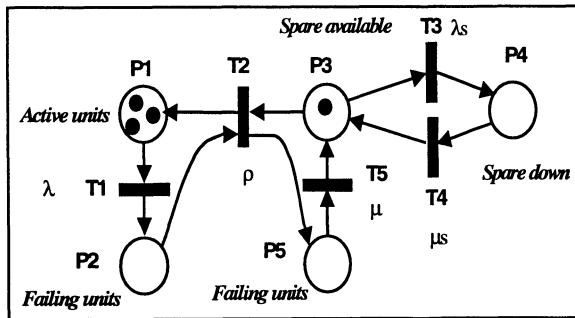


Figure E.8. Stochastic Petri net

The analysis of this graph can be performed by means of a finite state machine (non-parallel model), called the marking graph, which shows all possible evolutions from the initial state (3 tokens in $P1$ and 1 token in $P3$). We can notice that the total number of tokens is constant.

Example of evolution:

(P1=3, P3=1) - (P1=2, P2=1, P3=1) - (P1=3, P5=1, P3=0) - (P1=3, P3=1), etc.

Exercise 7.7. Fault Tree and Reliability Block Diagram

The fault tree can be analyzed with the knowledge of the reliabilities of the basic events (leaves of the tree). Hence, we start from the leaves and go up towards the studied event which is the failure of the system. The probability at the output of a AND node is the product of the probabilities at its inputs. The probability at the output of a OR node (here with 2 inputs) is the sum of the probabilities at its inputs minus the product of these probabilities (this can be generalized to a more complicated formula for *n* inputs). The failure of the system has the probability:

$$F = F12 + F3 - F12.F3 = (1-R1).(1-R2) + (1 - R3) - (1-R1).(1-R2).(1 - R3),$$

Hence, $R = 1 - F = (R1 + R2 - R1.R2).R3.$

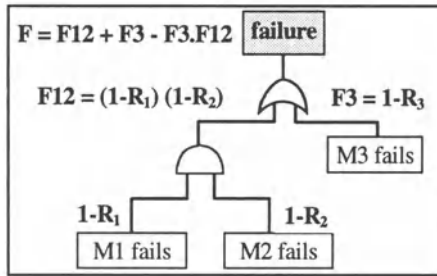


Figure E.9. Fault tree analysis

Figure E.10 shows the Reliability Block Diagram of this redundant system: two modules M1 and M2 in ‘parallel’, in ‘series’ with M3. The analysis by the method already studied gives the reliability: $R = R12 \cdot R3 = (1 - (1 - R1).(1 - R2)) \cdot R3 = (R1 + R2 - R1.R2).R3.$ We obtain the same result.

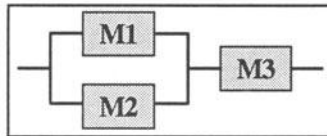


Figure E.10. Reliability Block Diagram of the system

Exercise 8.1. Functional redundancy of an adder

1. The number of input vectors producing a same output value is variable. This function increases from 1 to 10 according to a linear law when the output value varies from 0 (only one possibility: 0 + 0) to 9 (10 vectors: 0+9, 1+8, ..., 9+0). Then, it decreases from 10 to 1 when the output value passes from 9 to 18 (only one case: 9+9). Finally, it takes the constant value 0 when the output values are between 19 and 99, corresponding to ‘impossible’ cases. The input values being supposed as having the same occurrence probability, we deduce the probabilistic domain shows in Figure E.11.

$P(c) = (c + 1)/100$, for $c \in [0, 9]$, $P(c) = (19 - c)/100$, for $c \in [9, 18]$, and $P(c) = 0$ for $c > 18$.

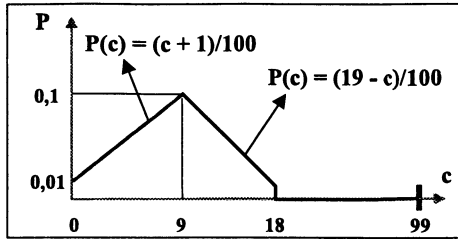


Figure E.11. Probabilistic static functional domain

Failure detection. A typical application of the functional redundancy deals with failure detection. Let us imagine an external observer receiving the output values. The preceding domains reveals that the output values belonging to (19, 99) are strictly impossible: if the observer receives a value belonging to this sub-domain, it can without any doubt signal the occurrence of a failure due to an unknown fault. If the output values belong to the acceptable sub-domain (between 0 and 18), this observer must record all produced output values and compare their occurrence rate with their probabilistic rate. In case of significant difference, this observer can raise a warning signal indicating that a failure might have occurred.

2. Input redundancy. There are $2 \times 4 = 8$ bits, i.e. 256 configurations, but only 100 of them are used by the considered code: 10 values for *A* and 10 values for *B*. The resulting input redundancy rate is: $(256 - 100) / 256 = 0.61$ (number of unused vectors divided by the total number of possible vectors).

Output redundancy. There are also 8 output bits, i.e. 256 configurations. Only 19 of them are used. So, the output redundancy rate is: $(256 - 19) / 256 = 0.93$.

3. The numbers have 2 bits and they are constrained by the property $A \leq B$. This leads to $1 + 2 + 3 + 4 = 10$ cases. Hence, the redundancy rate is: $(2^4 - 10) / 2^4 = 0.38$.

Exercise 8.2. Functional redundancy of a state graph

First of all, we observe that the graph has no redundant unreachable part: from the initial state 1, we can reach any other state. Then, we analyze the state graph to find if it accepts sequences that are forbidden by the input constraint: ‘*c* is never applied after *b*’. There are two such situations (see Figure E.12):

- a) The first one occurs when the graph is in state 2 or 3, if we apply the forbidden sub-sequence $\langle b, c \rangle$. As this situation will never occur, the arc 3-2 (by input *c*) is redundant and thus can be removed. Indeed, this transition is never fired by any acceptable sequence.
- b) The second one occurs when the graph is in state 4, and if we apply the same sub-sequence $\langle b, c \rangle$. However, the arc 4-2 (by input *c*) is not redundant. Indeed, the sequence $\langle a, a, c \rangle$ from state 1 is quite acceptable: it passes through states 2,

4, and finally reaches state 2 by the arc 4-2. In that case, the functional redundancy cannot be expressed as a redundant arc but as a redundant path: <4 - 4 - 2> is redundant.

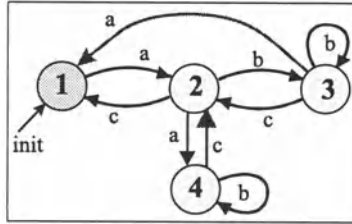


Figure E.12. Redundant graph

Exercise 8.3. Structural redundancy and faults

1. This circuit implements the logical functions: $f = a' + b$, $g = b'$. The table of Figure E.13 shows the resulting input/output configurations. Both faults considered here have no effect on f . Fault α has also no effect on g . Consequently, there is no failure. On the contrary, fault β is activated as a failure when $a.b = 10$: hence, the output f takes value '0' instead of '1'.

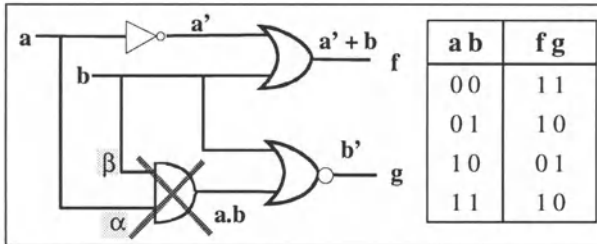


Figure E.13. Redundant circuit

2. We deduce from the previous study that the line α is totally superfluous. Thus, it corresponds to passive redundancy. Going a step further, the analysis shows that the output g is independent from the product term $a.b$ produced by the AND gate. Thus, this gate (noted X in the figure) can be removed, the NOR gate producing g being hence a simple INVERTER.
3. The truth table shows that the input configuration (00) never occurs at the output of the circuit: this corresponds to an *output functional redundancy*.

Exercise 8.4. Structural redundancy of several circuits

We suggest the following analysis to detect possible 'structural redundancies'. We establish the logical expression of each node of the circuit, starting from the primary inputs (the external inputs of the circuit) and going backwards to the primary outputs (the external outputs of the circuit). At each step, the resulting logical expression is analyzed in order to determine possible simplifications. If such simplifications exist, then they reveal structural redundancies.

Circuit 1. We determine $f = a + a.b' + c = a + b' + c$. Hence, the input line b of the AND gate is redundant. We can remove this line and also the AND gate, a entering directly into the NOR gate. This redundancy is *passive*.

Circuit 2. This circuit possesses passive structural redundancy: gate $(b + c)$ can be removed. No stuck-at 1 faults of this gate can be detected on the output f .

Circuit 3. This circuit realizes the majority function of its inputs without any structural redundancy: $f = a.b + a.c + b.c$.

Circuit 4. The input lines of circuit 4 are all different, excepted variable b which intervenes twice. The XOR being commutative, we can modify the network by shifting the terms b and $b.e$ to the beginning of this network. The function becomes: $f = b \oplus be \oplus ac \oplus d$. Now, $b \oplus be = be'$; hence, this circuit can be simplified. However, there is no passive redundancy: all stuck-at faults can be detected.

Exercise 8.5. Software redundancy and constraint types

1. The feature **new** creates a new type from another one (here the type 'integer'). Specific operations (subprograms) must be defined, as the ones provided by the other types cannot be used. For instance, two `Size_of_Shoes` cannot be added or divided.

The declaration

```
type Size_of_Shoes is new integer;
P: Size_of_Shoes;
instead of
P: integer;
```

is not a functional redundancy. The two versions lead to the same executable code which allocates one word in memory for the variable P .

On the contrary, it constitutes a *structural redundancy* of the source program. It is an *active redundancy* if subprograms using parameters of type `Size_of_Shoes` exist. Indeed, the associated operations are specific to `Size_of_Shoes` and not to any integers. On the contrary, this redundancy is *passive* if the program makes only use of integer operations.

2. The adding of the constraint

```
type Size_of_Shoes is new integer range 28..45;
```

reduces the number of acceptable values. For subprograms having parameters of this type, this declaration reduces the functional domains, hence having an impact on their functional redundancy. The type declaration itself constitutes a structural redundancy, as it corresponds to an element of the structure of the 'program model'. It seems to be passive, as its omission has no effect on the behavior of the system. The reality is more complex. If the structure of the program is such that no value outside the interval `[28..45]` can be attributed to P , the answer is 'yes': there is passive redundancy. On the contrary, if this hypothesis can be guaranteed, then the constraint cannot be removed from the declarative part, because the execution of the program leading to the assignment of a value outside the `[28..45]` range produces the raising of an exception (`Constraint_Error` in *Ada*), and thus a different behavior. In this case, the redundancy is active.

Exercise 8.6. Exception mechanisms of languages: termination model

The functional redundancy depends on the types of the parameters and on the actual values received and returned by the procedure. Let us signal that if the exception handler implements a full-tolerance, the returned values are the same, whatever an exception is raised or not during the body execution.

Structural redundancy exists. The exception handler is not useful if no exception (no error) occurs. So, the redundancy is passive. However, let us signal that if an exception is raised, it is then propagated in the software hierarchy when no handler exists. Hence, the handler removal changes the program behavior where errors are concerned.

This redundancy is *separable*: the handler is explicitly separated from the body. This feature thus constitutes an interesting means to show the normal body and the error handling part.

The redundancy is *off-line* (or inactive), as the handler starts its execution only when an exception is raised.

THIRD PART

Exercise 9.1. Requirement analysis

Two families of entities are defined in the text.

The first one concerns the capability of the product to be moved. This notion is specified by two entities: the product must be contained in a hand and the product must be moved by car.

The second family concerns the notion of autonomy specified as maximized.

Let us note that numerous specifications can be derived from these requirements. For instance, the autonomy can be provided by an efficient battery included in the mobile phone, and/or by a connection to the car battery.

Exercise 10.1. Verification of the adder

The functional fault considered transforms the adder into the circuit of *Figure E.14*.

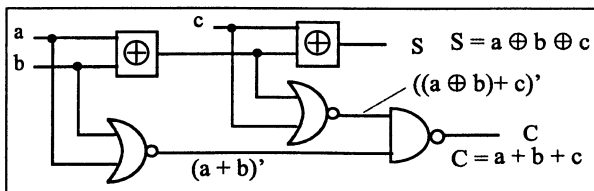


Figure E.14. Erroneous adder

1. Verification by extraction. We determine the new logical expressions of this circuit, starting from the primary inputs and progressing towards the primary outputs:

- $S = a \oplus b \oplus c$, the sum is not altered by the fault considered,
- $C = a + b + c$, the carry is erroneous (the correct function is $ab + ac + bc$).

There is a failure on output C each time one input only is at '1': so there are three erroneous vectors.

2. **Verification by double transformation with intermediate model.** We choose as intermediate model the modular description of the adder as two interconnected half-adders. The fault modifies each one of these half-adders: the behavior with and without fault is the same only when both inputs have identical values. The combination of these two modules is correct if and only if: $a = b = c = 0$ or $a \cdot b \cdot c = 1$. All others vectors give a wrong output.
3. **Verification by double top-down transformation.** The behavior of the circuit is simulated with a functional input sequence significant of the correct behavior. For example, we perform an addition without carry ($1 + 0 + 0$), and an addition giving the maximum output value ($1 + 1 + 1$). Then we compare the results given by the circuit with the computed theoretical values.

Exercise 10.2. Programming style (C language)

The four situations described hereafter stress examples of bad programming style, which increase the risk of introducing faults. Let us note that in each case, the program is syntactically and functionally correct. However, the bad style which is used makes it very probable to produce faults. Moreover, even for such simple functions, the style used for its specification will probably lead to utilization faults (calls to the function).

- The type of the returned value (`int`) is absent. This is syntactically correct (use of default type), but the user of this function may think that this function does not return any value. In a general way, the use of 'by default' or 'implicit' constructions is not at all advised.
- The name of the function is not explicit. In particular, the fact that it proposes two mutually exclusive treatments is not specified.
- The value '5' used in the specifications to define the size of the array is then reused in the loop! First, no link exists between these two values dealing with the same constant (same concept). Moreover, the maintenance operations can introduce faults if the size of this array is modified. It is much preferable to explicitly define a constant by means of a `#define` before the function.
- The parameter B has a non-explicit name; moreover, the associated type (`int`) reinforces this ambiguity. One always must explicitly define the Boolean type by an enumerated type or by defining the two constants `TRUE` and `FALSE`.

Exercise 10.3. FSM synthesis

The reasoning has been presented in section 10.4 of Chapter 10 dealing with functional testing. It is based on a simulation of the automata, in order to compose them as one automaton. This composition process reveals the input/output relations and removes the internal interactions between the automata (these relations have been introduced by the design process).

Exercise 10.4. Functional test sequence

In order to obtain a functional test sequence of the drinks distributor, we develop a three-step procedure:

- formal specification of the behavior,
- definition of an input sequence which provokes the complete activation of this behavior,
- deduction of the expected output values in response to the preceding input sequence.

The formal specification is derived from the informal specification. It describes two aspects shown in *Figure E.15*: the interface and the behavior.

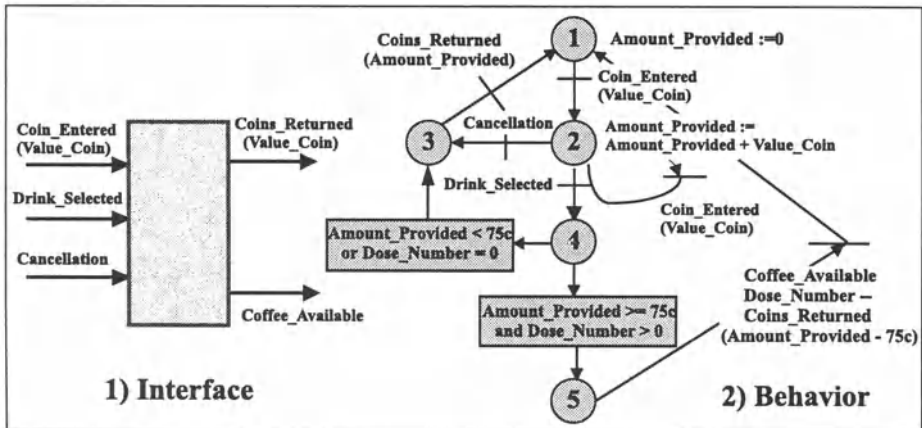


Figure E.15. Interface and behavior

Functional modeling is made by an automaton which is well adapted to the sequential features of this system. We have added to this automaton some annotations dealing with the data treatments (e.g. the addition operations, etc.). Let us note that this model describes only functional aspects of the specification. Other points, such as those dealing with the ergonomics of the system, are not expressed.

The formal modeling of the specifications for test generation purpose is interesting for two main reasons:

- Rules allowing to deduce what are ‘all possible behaviors’ can be defined for each modeling means. For example, in the case of an automaton, one might want to pass through each state or else through each transition between states. This aspect is developed in Chapter 13.
- The application of the rules can be systematic; that is to say, we are able to deduce a sequence activating all behaviors (in the sense of the rules). Tools can then automate the production of the test sequences.

In our case, we will make this work ‘by hand’, with the following assumptions:

- every path must be exercised at least once by the sequence,
- if an arc is conditioned by a Boolean expression, we must pass through that arc:
 - with one internal value belonging to each *domain* defined by this expression,
 - and with the limit values between these domains.

At first, we define the set of all paths, we give a name to each path, and we enumerate its states.

- Enter a coin and cancel: {1, 2, 3, 1}.
- Enter two coins and cancel: {1, 2, 2, 3, 1}. The presence of a loop from state 2 introduces an infinite number of paths; we limit the number of iterations to 1.
- Order a coffee after having entered a sufficient number of coins, if the number of doses is greater or equal to 1: {1, 2, 4, 5, 1}. The condition labeling the arc (4, 5) induces a domain of values for the couple (Amount_Provided, Dose_Number). It is necessary to take a value $\geq 75c$ (for example 1\$) and a number of doses > 0 (for example 2). Moreover, we must apply 'limit tests', i.e. $75c$ and 2 doses, then 1\$ and 1 dose. Consequently, 3 sequences must be defined for this path. This situation shows also an interesting aspect dealing with the memory implied by the used variables. Indeed, when we apply the first part of the sequence {1, 2, 3, 1}, the expected behavior is the same, whatever the past of the system. Moreover, this behavior will have no effect on the future. On the contrary, in order to test the behavior of the system when only one dose remains, it is necessary to first apply sequences leading the system in the required initial state (one dose only); these sequences are called *initialization* or *homing sequences*. Finally, once the test consuming the last coffee dose has been performed, it will be necessary to continue the test procedure with test sequences, assuming a null number of doses. To conclude, the various test sequences we have defined are not independent; hence, these fragments must be scheduled in a coherent order, maybe with extra link sub-sequences.
- Case where the user orders one coffee after entering a sufficient number of coins, but when there are no more coffee doses: {1, 2, 4, 3, 1}. As said before, the preceding parts of the global test sequence must have led to a situation where no coffee doses remain. Otherwise, the condition labeling the arc (4, 3) being constituted by a Boolean expression using one OR, it is also necessary to test the opposite situation, e.g. 'Amount_Provided < 75c', and the two simultaneous situations, i.e. 'Amount_Provided < 75c and Dose_Number = 0'.

From this analysis, we can deduce the various pieces of sequences associated with each tested fragment of behavior. *Table E.5* gives an example for the first case considered. We will not develop the whole set of test sequences. Its obtaining is easy as far as we take care of the necessary relationships between those fragments, according to the state of the system. This job may seem to be tedious. However, it is systematic, providing a good guarantee that the resulting functional test sequence activates properly the whole set of possible behaviors.

Input	Output
Coin_Entered (50c)	
Cancellation	
	Coins_Returned (X\$)

Table E.5. Sequence

Exercise 10.5. Property research

In the previous Exercise, the need of the client is to earn money. One can ask the question: "Is it possible to get a coffee without paying or with less than 75c?"

This analysis must first be conducted on the specification model. Here the answer is 'no', as *Coffee_Available* is conditioned by '*Amount_Provided* \geq 75c' (because of the OR function). On the contrary, one must then ask himself if *Amount_Provided* effectively contains the amount of money which has been entered in the machine. It would not be the case if *Amount_Provided* were initialized to 1\$, and then not a signed in the model. The analysis of the backward path shows that the amount of entered coins is effectively the value of *Amount_Provided* in state 4.

Exercise 10.6. Properties of functional graphs

Independently from any functional aspect of the product (we ignore its function), we try to define properties which are significant of the studied functional graph.

1. If state 4 is suppressed, the graph is split into two independent sub-graphs; hence, it is no more possible to pass from one sub-graph to the other. Let us note that each sub-graph is alive. This situation can correspond to two independent modules. The global functioning results from the 'Cartesian product' of these two sub-graphs: every state couple from the two graphs is theoretically possible.
2. On the contrary, if we add an arc joining state 5 to state 1 (in bold in *Figure E.16*), the connectivity of the graph is increased. It is then possible to draw a table containing all the states reachable from any state:

from state 1 or 2 or 3, one can reach states 1, 2, 3, 4; from state 4, only state 4 can be reached; from state 5 or 6, one can reach states 1, 2, 3, 4, 5, 6.

State 4 remains a state which definitely blocks the functioning of the system: hence, this situation corresponds to a locking.

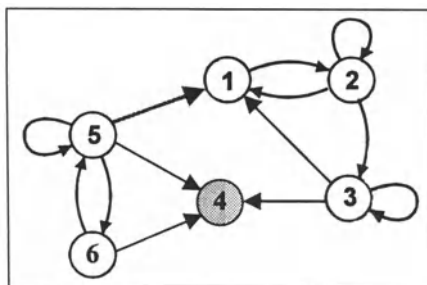


Figure E.16. Graph

Exercise 10.7. Verification of a floating-point unit

Black box verification by functional simulation. We are looking for a simulation sequence which passes through each module of the product and activates a maximum of functions and connections. A simple sequence would include a series of additions on several numbers:

$$A1 = M1 \cdot 10^{E1} \text{ and } A2 = M2 \cdot 10^{E2}.$$

We must check the module performing the subtraction ($E1 - E2$) with different exponents: ($E1 > E2$), then ($E1 < E2$), positive values, then negative values. These operations also verify the circuit which performs the ‘adjust’ operation (right shift of the mantissas).

Then, we must check the circuit calculating the final ‘sign’ of the result. For this purpose, we make several ‘+’ and ‘-’ operations with numbers having the same sign, and finally opposite signs. The sign S of the final result must take into account the carry coming from the +/- circuit. Hence, we consider a situation such that $M'1 > M'2$ for an adding control (signal +/-): e.g. subtraction of two negative numbers, the absolute value of the subtracted one being greater than the first one. If the result of the circuit ‘+/-’ is greater than 1, there is a carry, and we must perform a normalization operation, i.e. add ‘1’ to the exponent and make a one-figure shift to the right of the mantissa.

Finally, the overflow situations must be considered. For example, we add two negative numbers with maximum value positive exponents (+999 if E is expressed with 3 digits), and such that $|M1| + |M2| \geq 1$.

Exercise 10.8. Inductive formal proof

1. We must demonstrate that $A1 \implies A2$ when $R \geq B$ after the execution of $R := A$ and $Q := 0$. The second condition of $A2$ is evident: it is the loop assertion($R \geq B$). As $R := A$ and $Q := 0$, then $Q*B + R = 0*B + A = A$. So, the first condition of $A2$ is true.
2. We must demonstrate that $A1 \implies A3$ when $R \geq B$ is false after the execution of $R := A$ and $Q := 0$. The condition ‘ $R \geq B$ is false’ implies that $R < B$. We have $Q*B + R = 0*B + A = A$. So, the first condition of $A3$ is also true.
3. We must demonstrate that, when [$A2$ is true and $R := R - B$ and $Q := Q + 1$ are executed and then $R \geq B$], then $A2$ is true with the new values of R and Q . Let us note Rb and Qb the values of R and Q before the execution. The hypotheses are $A = Qb*B + Rb$ (relation 1), and $Rb \geq B$ (relation 2). After execution of the loop statements, we obtain $R = Rb - B$ (relation 3) and $Q = Qb + 1$ (relation 4). The relation $R \geq B$ is true due to the loop condition. We must demonstrate that $A = Q*B + R$. Relations 3 and 4 give: $Q*B + R = (Qb + 1)*B + (Rb - B) = Qb*B + B + Rb - B = Qb + B + Rb = A$ (relation 1). So the second condition is demonstrated.
4. The demonstration of $A2 \implies A3$ after the execution of the loop statements and when condition $R \geq B$ is false is quite similar concerning the second condition. The second condition $R \geq B$ is due to the negation of the loop condition.

Exercise 11.1. Component choice

Failure rate of the first structure. The failure rate is the sum of the failure rates of the components (as these values are very small: this would not be true otherwise!):

$$\lambda_1 = 12.10^{-7} + 1.10^{-6} + 3.10^{-5} = 3.22.10^{-5}.$$

Failure rate of the second structure: $\lambda_2 = 4.10^{-6}$.

Thus, the second structure has a better reliability than the first one.

Note. This exercise does not consider the influence of temperature or radiations on the reliability of these components, or their mutual influence.

Exercise 11.2. Comparison of the reliability of two products

The two failure rates λ_1 and λ_2 evolve according to power of 10. Hence, $\log_{10}(\lambda_1)$ and $\log_{10}(\lambda_2)$ are linear. We deduce from this the two logarithmic equations for the two products:

1) For λ_1 : $\log_{10}(\lambda_1(T)) = [\log_{10}(10 \cdot \lambda_{01}) - \log_{10}(\lambda_{01})]T / (38-18) + b$, where $\lambda_{01} = \lambda_1(18^\circ)$.

When $T = 18^\circ\text{C}$, we have $b = -59/10$.

So, $\log_{10}(\lambda_1(T)) = T/20 - 59/10$.

2) The same reasoning for λ_2 gives $\log_{10}(\lambda_2(T)) = T/10 - 88/10$.

Finally, we deduce T for $\lambda_1 = \lambda_2$: from 58°C the reliability of $P1$ becomes better than the reliability of $P2$.

Exercise 11.3. Shared FIFO

1. The result is hazardous, as the data structure (array and indexes) is shared (same situation as in sub-section 11.2.2.2). Moreover, the data structure value may be incoherent. For instance, consider the scheduling described in *Figure E.17*, where WI expresses the variable `Write_Index`.

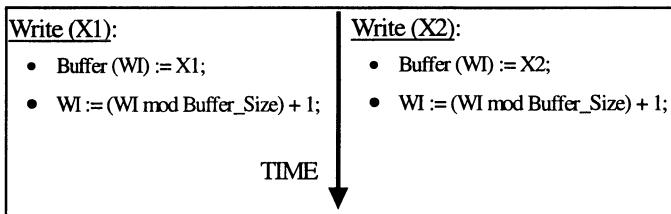


Figure E.17. Incoherent value of the array

After execution of this sequence of statements, only $X2$ is memorized, as it overloaded $X1$, and an empty item is created, as `Write_Index` is incremented two times. Such a situation defines an error, that is to say an unacceptable state of the data structure value.

2. No problems occur if we consider that the list is never empty, as the two couples of data structures (`Write_Index`, `Buffer (Write_Index)`) and (`Read_Index`, `Buffer (Read_Index)`) have no common elements. However, no mechanisms guarantee that a reading cannot occur when the FIFO list is empty.

To conclude, this implementation induces a high risk of error occurrences. The problems come from a characteristic of the task management implementation: the preemption of a task by another task does not guarantee exclusive access to the

shared resources. To be safe, a mechanism managing the access authorizations (such as a semaphore) must be added.

Exercise 11.4. Hazards in shared variable implementation

No problems seem to exist in the sharing of variable using one statement. Indeed, the design is correct. However, assume that the incrementation and decrementation operations are processed on a register AX. Then, I^+ is translated as Task1:

```
Move AX, @I
Inc AX
Move @I, AX
```

In the same way, the statement I^- is translated as a Task2:

```
Move AX, @I
Dec AX
Move @I, AX
```

Consequently, several instructions are necessary to implement one statement. *Figure E.18* shows the two considered implementations of these tasks: sequential or interleaved.

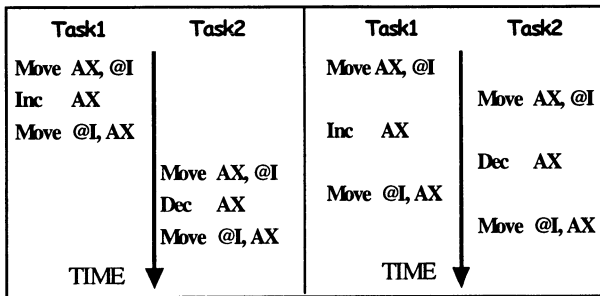


Figure E.18. Hazardous result

Let us show that the result is hazardous. On the left side of *Figure E.18*, the result is unchanged at the end of the execution of Task1 then Task2. This is the expected result, as the value of I is incremented and decremented. On the contrary, on the right side of *Figure E.18*, each task saves and restores its own context (in a local Task Control Block) at each task switch. After the execution of the second line of Task1 (`Inc AX`), this value is not transmitted to Task2 that decreases its own copy of AX when executing its second line. Thus, the final value of I is incorrect.

Exercise 12.1. Signature testing

1. The sequential treatment of the binary flow comprises 64 (i.e. $1024 / 16$) XOR operations on consecutive 16-bit words. If we suppose that the signature of the faultless circuit is known, any multiple error altering one or several words is detectable if and only if any modified bit of a word is also modified an odd number of times in the same position of several words. According to the output stream, this corresponds to erroneous bits repeated an odd number of times

modulo 64. For example, a multiple error altering bits 1, 15, 65, 121 is detectable: errors 1 and 65 neutralize themselves, but each error 15 and 121 is detectable. All functional or technological faults producing such errors are thus detected. All other faults are undetectable.

- Without any knowledge about the electronics implementation of this system (gate or MOS structure), one cannot deduce any class of technological faults that produces the preceding errors.

Exercise 12.2. Toggle test sequence

A Toggle Sequence is such that every line in the circuit takes the values ‘0’ and ‘1’:

- each XOR must receive a vector from the set {00, 11}, and a vector from the set {01, 10},
- each NAND must receive the vector (11), and, either the two vectors (01) and (10), or the vector (00).

We propose the following Toggle sequence: <010, 101, 111> (there are other solutions). The reader will verify that this sequence applies ‘0’ and ‘1’ to each line.

Exercise 12.3. Test of components

- Statistically speaking, $y\%$ of the products are good and are tested with a duration of n ‘time units’. The faulty products correspond to $(1 - y)\%$ of the production. As the test coverage is $c = 80\%$, 20% of these faulty products, that is to say $(1 - c).(1 - y)$ of the total production, will be considered as good by the test sequence after a duration of n time units. Finally, $c.(1 - y)$ products will be declared as wrong after a mean duration time of $n/2$ time units.

Therefore, the mean time dedicated to the test of this production is:

$$d = [n \cdot y + n/2 \cdot c.(1 - y) + n \cdot (1 - c).(1 - y)] = [n \cdot (1 - c / 2 + c.y / 2)] = 0.996 n.$$

- The rate of faulty products not detected as faulty is: $(1 - c).(1 - y) = 2\%$.
- If only 70% of the products are submitted to test, the mean time is reduced.

The ratio of non-detected faulty products has two terms:

$$t.(1 - c).(1 - y) + (1 - t).(1 - y).$$

The first term corresponds to faulty products which are tested but not detected as faulty; the second one corresponds to faulty products which are not tested.

Numerical value: 4.4% .

Exercise 12.4. Fault coverage

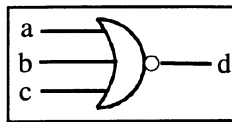


Figure E.19. A 3-input NOR gate

The NOR gate is symmetrical, according to its inputs *a*, *b* and *c* (Figure E.19). Consequently, the study can be reduced to the case of one input only, e.g. *a*. The other inputs (*b* and *c*) must be set to the value '0' in order to let the error pass to the output. If an input is set to the value '1', it forces the output to the value '0'.

1. **Optimal test sequence.** There is only one optimal test sequence comprising 4 test vectors: <000, 100, 010, 001>.
2. **Coverage.** Each input vector covers some stuck-at faults of the I/O lines. Table E.6 shows the fault coverage of each input vector.

We notice that some input vectors have a very small coverage; they should not be taken to test this circuit; thus, (011), (101), (110), and (111) test the stuck-at 1 of line *d* only. On the contrary, the vector (000) covers half of stuck-at 0/1 faults.

Input vectors	Test coverage			
	a	b	c	d
0 0 0	1	1	1	0
0 0 1	-	-	0	1
0 1 0	-	0	-	1
0 1 1	-	-	-	1
1 0 0	0	-	-	1
1 0 1	-	-	-	1
1 1 0	-	-	-	1
1 1 1	-	-	-	1

Table E.6. Fault coverage

Figure E.20 shows the coverage curves of: 1) the exhaustive sequence, 2) the optimal sequence, and 3) the very simple toggle test sequence <000, 111>.

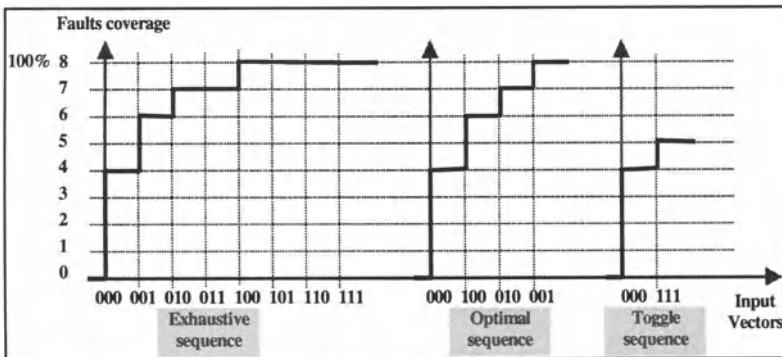


Figure E.20. Fault coverage evolution

Exercise 12.5. Simple fault diagnosis

Let us analyze the coverage table obtained in the preceding exercise.

We observe that all stuck-at 0 faults are detected by distinct test vectors. Hence, they can all be distinguished by the sequence <000, 001, 010, 100>. All these faults can also be distinguished from the stuck-at 1 fault of the output. We also observe that all stuck-at 1 faults of the inputs and the stuck-at 0 of the output are detected by the vector (000) only. Consequently, they cannot be distinguished from the outside. Hence, they are said to be *equivalent*.

Exercise 12.6. Optimal test sequence

Figure E.21 reminds the gate structure of the circuit. Input vector 1 (respectively 2) apply 11 to gate A (respectively B), and 10 (respectively 01) to gate C. (see Table E.7). Hence, any stuck-at 0 fault is detected: activated as an error, and the error propagated to *f*. Let us note that the input vector 111 would apply 11 simultaneously to gates A and B, but no stuck-at 0 of these gate would be observable on *f*. Let us also note that when receives 11, B (or A) receives one of the vectors 10 or 01; unfortunately, these vectors cannot be ‘counted’ as belonging to the minimal AND test sequence, because gate C will not propagate any error coming from B (or A).

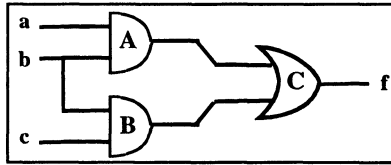


Figure E.21. AND-OR circuit

Hence, input vectors 3 and 4 are necessary to apply the missing configurations: 01 and 10 to the AND gates, and 00 to the OR gate. These vectors will detect all the stuck-at 1 faults: activation as an erroneous ‘1’ error, and propagation of this erroneous ‘1’ to the output. The optimal test sequence has 4 vectors: <110, 011, 010, 101>.

Inputs a b c	Gates		
	A	B	C
vectors	01 10 11	01 10 11	00 01 10
1 1 1 0	X		X
2 0 1 1		X	X
3 0 1 0	X	X	X
4 1 0 1	X	X	X

Table E.7. Optimal test sequence

Exercise 12.7 Sequential circuit testing

1. Table E.8 shows the evolution of the sequential system submitted to the input sequences *ST1* and *ST2*, applied to the same initial state 1.
2. A simple simulation of the logical circuit allows establishing the different values of each node for test sequences *ST1* and *ST2*. Then, we deduce lines that take both values 0 and 1.

	ST1	ST2
e	0 1 1 0	0 1 1 0 1 1 0 0 1
q	2 4 3 1	2 4 3 1 2 4 2 3 1
s	1 0 1 0	1 0 1 0 1 0 1 1 0

Table E.8. Correct and erroneous functions

3. Complete structural test.

- First, we determine those faults which are not detected by the functional test sequence *ST2*. This step can be performed thanks to a ‘fault simulator’ such as *Verifault* of *Cadence*. Thus we can identify the 5 stuck-at faults which are not detected, reducing to 3 classes of equivalent faults.
 - Now, if we want to test one of these remaining faults, we can proceed as for combinational circuits. We first try to activate this fault by setting the faulty line at the opposite value. Then, a backward procedure is applied. Because of the feedback loops, this procedure generally does not easily converge towards a solution. The fault is detected when it is transformed into an erroneous output value. The Reset input can be useful to perform this procedure, assuming it has previously been tested.
4. The problem of the initialization of a circuit prior to the application of a testing sequence is not always easy. Generally, it is assumed that there is a reset input which switches all flip-flops to the zero state. In the very general case of asynchronous sequential systems without such inputs, it is necessary to find special initialization input sequences called *homing sequences*. Initialization is a real problem for testing complex systems.

Exercise 13.1. Test of a small circuit

1. **Fault Table.** Faults detected by the different input vectors can be obtained from the logical circuit of *Figure E.22*, by applying the methods studied in Chapter 13, either column after column, or row after row.

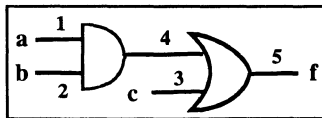


Figure E.22. Two-gate circuit

Table E.9 shows the results obtained. There are three ‘best test vector’: 010, 100 and 110. Each one covers 4 faults. The input vector having the lowest coverage is 111 with only 1 fault detected.

2. **Minimal test sequence.** Some faults are detected by one test vector only; it is the case of faults 1^1 (vector 010), 1^0 (vector 110), 2^1 (vector 100), and 2^0 (vector 110). Hence, the 3 vectors 010, 100 and 110 belong to any minimal test sequence. There are three minimal-length test sequences, for example $TS = \langle 001, 010, 100, 110 \rangle$.

a b c	1	2	3	4	5
0 0 0	-	-	1	1	1
0 0 1	-	-	0	-	0
0 1 0	1	-	1	1	1
0 1 1	-	-	0	-	0
1 0 0	-	1	1	1	1
1 0 1	-	-	0	-	0
1 1 0	0	0	-	0	0
1 1 1	-	-	-	-	0

Table E.9. Fault table

Exercise 13.2. Test vectors detecting a fault

The *fault activation* condition implies to put a ‘1’ value to line 11. By *backward propagation* of this condition, we find three possible cases on lines 5 and 6, i.e. $bc = 10, 01$ and 11 . The *forward propagation* of the error produced on α can follow two different paths: $P1$ (through lines 10 - 13 - f), and $P2$ (through lines 15 - g).

These two paths will never simultaneously conduct errors to the outputs f and g .

Path $P1$. The local propagation conditions are $9 = 0$ and $12 = 1$.

We find the 5 following test vectors ($a b c d$): $(0 1 0 -)$, $(0 1 1 0)$, $(- 0 1 0)$.

The resulting failure on output f is a ‘1’ value instead of a ‘0’ value without fault.

Path $P2$. The local propagation condition is $16 = 0$.

So we must have $7 = 8 = 1$. We find the 4 following test vectors: $(a b c d) = (- - 1 1)$.

The resulting failure on output g is a value ‘0’ instead of a value ‘1’ without fault.

Summary: there are 9 test vectors for this fault, 5 of which produce a failure on f (0100, 0101, 0110, 0010, 1010), and the 4 other ones produce a failure on g (0011, 0111, 1011, 1111).

Exercise 13.3. Analysis of test procedures

1. This procedure explores the input space with a partition technique. The first objective is to activate the fault. A tree is built, each branch corresponding to a disjointed input cube (sub-set expressed with ‘0’, ‘1’ and ‘x’ values). The simulation propagates the known values towards the fault location (*Figure E.23*).

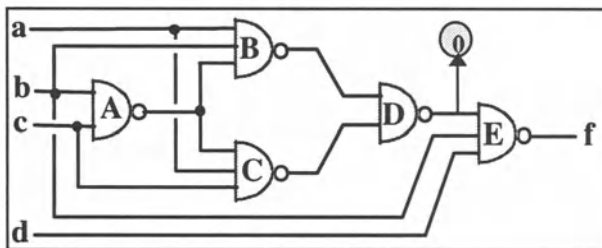


Figure E.23. NAND-gate circuit

Then, the fault is propagated to the output with the same exploration procedure until a solution is found (if any test vector exists). Obviously, this very simple technique can be long to converge towards a solution if the first possible test vector has many '1' values for a, b, c , etc.

Let us now complete the given procedure:

- Now, the objective is to propagate the error through gate E . A propagation towards gate E of the known values is performed (if it has not already be done!): $E = 0$, so the error cannot be transformed into a failure on output f . We make a backtracking in the input assignment.
- Input c is set to 'x', and input b is switched to '1', and a propagation is performed: the fault remains passive.
- Input c is set to '0', and a propagation is performed: $A = 1, B = 0, C = 1$, hence the fault is activated.
- Now, the objective is to propagate the error through gate E . Input d is set to '0', which forces output f to '1': this is a case of inconsistency, so we go backwards.
- Input d is switched to '1', and the error is finally propagated to f .

Thus, we obtain the test vector: $(a\ b\ c\ d) = (1\ 1\ 0\ 1)$.

2. This procedure makes a backward propagation along one path from the fault activation. At a given gate, if several vectors satisfy the desired output value, we choose the easiest path only (the closest path to the primary inputs). If several inputs must be set (to '0' or '1'), we choose the hardest path first (having the higher number of gates to the primary inputs). As usual, when the fault is activated as an error, we try to propagate the error along a path. All the process uses a backtracking technique in case of inconsistency. This method is close to the PODEM algorithm.

- Input b is switched to 1, and we perform a propagation action: it brings nothing.
- Input c is set to '0', and we perform a propagation: $A = 1, C = 0$, so the fault is activated.
- Now, the objective is to propagate the error through gate E . Input d is set to '1'.

We obtain the same test vector $(a\ b\ c\ d) = (1\ 1\ 0\ 1)$. It is the only vector which detects the fault.

Let us note that this procedure is not pertinent for this circuit. Indeed, at step 5 we have chosen to set b to '0' in order to force $A = 1$; this led to an inconsistency. Then, we have abandoned this path to try another one. Instead, at this point, we can try the second way to have $A = 1$, which is to set c to '0'. Then, the procedure sets a and b to '1'. Thus, the test vector is rapidly found.

Exercise 13.4. Fault coverage of a test vector

1. The structural analysis of the circuit gives the fault detection table shown in *Table E.10*): detection at output f , and at output g .

Note about 'reconvergent fanout' structures. We observe that the stuck-at '1' of line 2 is not detected on output f : this fault produces an error on line 9 and an opposite error on line 10, these errors being neutralized by gate 13. We also note that the stuck-at 1 of line 3 is detected on output f : it produces two identical errors on lines

13 and 15, which propagate through the output gate giving f . This same fault is also detected on output g : it produces two identical errors on lines 15 and 16, which propagate to the output g .

abcd	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1001 detection on f	-	-	1	-	1	1	-	-	0	0	0	0	0	0	0	0	0	0
detection on g	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1

Table E.10. Fault detection

- This test vector covers 11 of the 36 possible faults. The theoretical maximum coverage of a test vector is 18. Now, we can try to find the best test vector by analyzing the structure of the circuit. We know that the best test of AND, OR, NAND and NOR gates is obtained when their inputs take the neutral element value. The worst case is when all their inputs take the opposite values. The circuit we consider is made of a mix of AND, NAND, OR, and NOR gates; it is easy to see that no input vector will apply the optimal configuration to each gate. So no test vector will covers 50% of the faults! A good exploitation of all these local constraints is given by the input vector (0 1 0 1) which covers 13 faults!

Exercise 13.5. Diagnosis of a circuit

- Faults 2^0 and 5^0 are activated by the same constraint: $b = 1$. Fault 11^1 is activated by $b = 0$, or by $b = 0$ and $c = 1$. Hence, we can separate these two groups of faults by applying test vectors with $b = 1$, and test vectors with $bc = 01$. A first test vector could be $(a b c d) = (- 0 1 1)$ which detects fault 11^1 at output g . Now, we must try to distinguish faults 2^0 and 5^0 . Fault 2^0 can only be detected on f by applying the input vector (1110). Faults 2^0 and 5^0 can be detected on f (through the path 11 - 10 - 13 - 17) by the input vectors (010-). So, here is an example of diagnosis sequence: $DS = \langle 0011, 0100, 1110 \rangle$.

The related fault tree is drawn in *Figure E.24*. It gives all information necessary to diagnose one of these faults. For instance, if the *signature* corresponding to the application of the test sequence is $\langle OK, f KO, f KO \rangle$, then the identified fault is 2^0 .

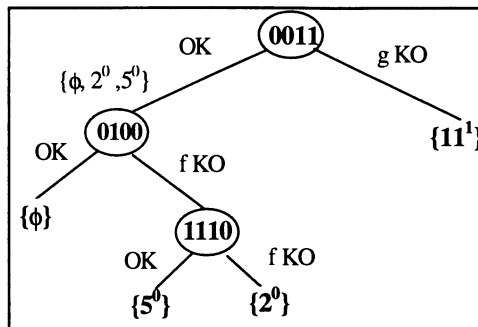


Figure E.24. Fault tree

2. We first determine all faults detected by vectors (1000), (1001), and (0110) on outputs f , g and both. This step is achieved by using the backward analysis method presented in Chapter 13. We obtain the partial fault table of Table E.11. Note that some faults are detected on f only, on g only, or on both outputs.

From this table, we deduce the fault tree (Figure E.25) corresponding to the test sequence: $TS = \langle 1000, 1001, 0110 \rangle$.

T	a	b	c	d	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	
T1	1	0	0	0	f	-	-	1	-	1	1	-	-	-	0	0	0	0	0	-	-	0	-
					g																		1
T2	1	0	0	1	f	-	-	1	-	1	1	1	-	-	0	-	0	0	0	-	0	0	-
					g			1															1
T3	0	1	1	0	f	1	-	-	-	-	-	-	1	1	1	-	1	-	-	-	1	-	-
					g							1				0					0		1

Table E.11. Partial fault table

3. The diagnosis power of this sequence is not good. Many faults belonging to the resulting fault classes can easily be distinguished by adding other test vectors.

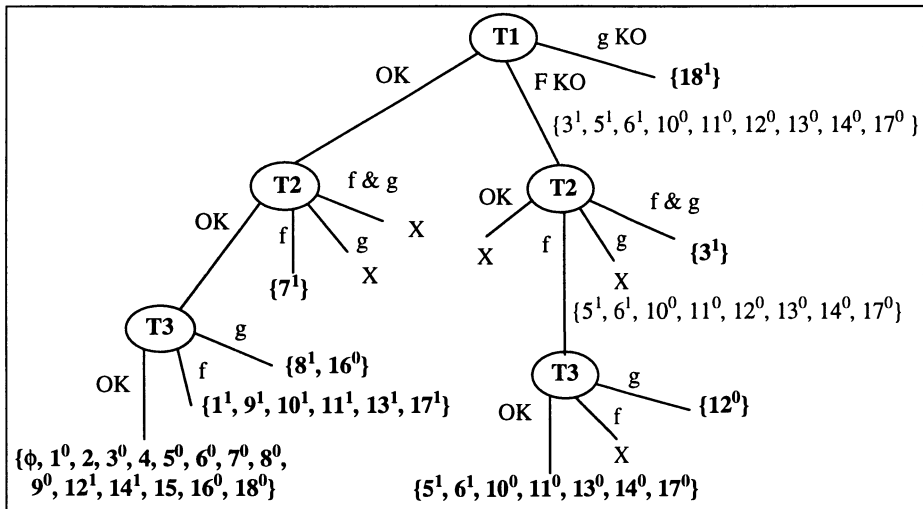


Figure E.25. Fault tree

Exercise 13.6. Complete diagnosis of a small circuit

The minimal test sequence obtained in question 2) of Exercise 13.1 is also a minimal diagnosis sequence: $\langle 001, 010, 100, 110 \rangle$. This sequence separates the following classes: $\{\phi\}$, $\{1^1\}$, $\{2^1\}$, $\{3^0\}$, $\{5^0\}$, $\{1^0, 2^0, 4^0\}$, $\{3^1, 4^1, 5^1\}$.

All faults belonging to these groups are equivalent. They cannot be distinguished from the outside

Exercise 13.7. Logical test of a full-adder

1. To activate this fault, we must set input b to '1'. This initial error (noted e in Figure E.26) is propagated to output S , for any values of inputs a and c : hence, we obtain 4 test vectors $(a b c) = (- 1 -)$ according to the output S .

This initial error can also be observed on output C if it can be propagated through the two NAND gates (see the figure). For this purpose, we must have $a = 0$ and $c = 1$. It leads to the test vector: $(a b c) = (0 1 1)$.

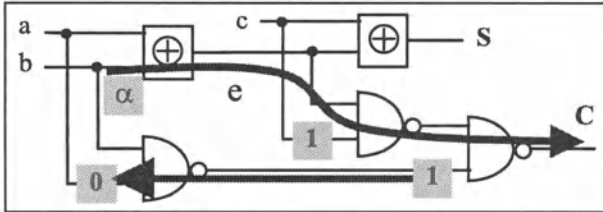


Figure E.26. Test of the half-adder

2. In Exercise 5.2, we have found all the failure configurations by a different approach: functional extraction, then comparison with the truth tables.
3. Any input vector detects every functional or physical fault that modifies the output value. Hence, it is not surprising if a vector obtained in question 1) is able to detect the functional fault consisting in transforming the XOR gates into IDENTITY gates.

Let us analyze the fault on the circuit's structure. Any vector $(a b)$ will provoke an error at the output of the first IDENTITY gate; this error (noted e in the previous figure) will again be transformed through the second IDENTITY gate and produce a correct output value. Thus, the fault is not observed on output S (we have already proved this property by *extraction* in Exercise 10.1). The conditions necessary to propagate error e towards the carry output C are exactly the same as for fault α of question 1). Hence, the functional fault is detected by the input vector $(a b c) = (0 1 1)$ which also detects fault α .

Exercise 13.8. Functional and toggle test of a full-adder

The structure of the circuit is given in Figure E.27.

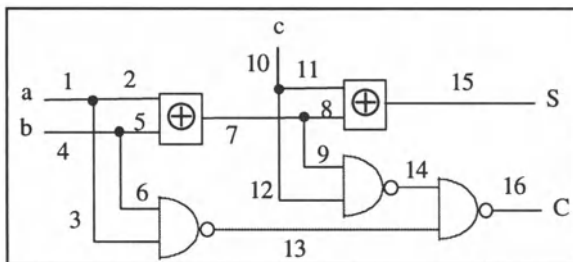


Figure E.27. Logical structure of the adder

1. **Function test sequence.** A very simple functional sequence will make one addition with $(SC) = (00)$, and one addition with $(SC) = (11)$. This sequence is $TS1 = \langle 000, 111 \rangle$. The two first lines of *Table E.12* show faults detected by this sequence. These faults have been determined by the structural method proposed in Chapter 13 applied to the logical structure (*Figure E.27*).

Test Sequence	a b c	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Functional	000	1	1	-	1	1	-	1	1	-	1	1	-	0	0	1	1
	111	0	0	0	0	0	0	1	1	-	0	0	-	1	-	0	0
Toggle	101	0	0	-	1	1	-	0	0	0	0	0	0	-	1	1	0
	001	1	1	-	1	1	-	1	1	1	0	0	-	0	0	0	1
	010	1	1	1	0	0	-	0	0	-	1	1	1	0	0	0	1
Complete	100	0	0	-	1	1	-	0	0	0	0	0	0	-	1	1	0

Table E.12. Test coverage

2. **Toggle test.** In order that each line takes values '0' and '1', we add the test vector 101; hence, the sequence becomes: $TS2 = \langle 000, 111, 101 \rangle$. The third line of the table shows faults detected by this new vector. We observe that $TS2$ does not detect 4 faults, confirming the fact that a toggle test is generally not sufficient to test every stuck-at fault.
3. **Complete test sequence.** Faults not detected by sequence $TS2$ are $3^1, 6^1, 9^1$ and 12^1 . To detect these faults, we must add three test vectors to $TS2$: 001, 010 and 100. The resulting complete test sequence has then 6 vectors: $TS3 = \langle 000, 111, 101, 001, 010, 100 \rangle$. This complete test sequence is not optimal in terms of number of test vectors. An optimal sequence, is made of 5 test vectors, such as: $TSop = \langle 001, 010, 100, 110, 101 \rangle$.

Exercise 13.9. Test of a structured circuit

1. Test of the structured adder. We will see that it is possible to apply the given complete test sequence to each module.

Test of module FA1. The controllability of this module is complete, so the five test vectors can be applied. The observability of output $S1$ is also complete. The only problem that remains is the observability of signal $c1$. Any error on this line will change the parity of the inputs of module $M2$; hence, this error is propagated to the primary output $S2$. Consequently, the first full-adder is completely tested.

Test of module FA2. Here, the only problem to analyze is the controllability of line $c1$. Whatever we put on inputs $a2$ and $b2$, it is easy to bring either a value '0' on line $c1$ (no carry for bits $c0, a0$ and $b0$) or a value '1' on line $c1$ (by producing a carry for bits $c0, a0$ and $b0$).

2. Test sequence. A complete test sequence of 5 input vectors ($c0, a1, b1, a2, b2$) can be obtained: $TS = \langle 00101, 01010, 10011, 11000, 01110 \rangle$

In conclusion, this structured circuit is easy to test. Naturally, one should not deduce that all structured circuits are easy to test.

Exercise 13.10. Diagnosis study of the full-adder

We draw first the partial fault table indicating, for each input vector, the outputs where the faults are detected (*Table E.13*).

abc	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
000 S	1	1	-	1	1	-	1	1	-	1	1	-	0	0	1	-
C	-	-	-	-	-	-	-	-	-	-	-	-	0	0	-	1
110 S	1	1	0	0	-	0	0	1	1	1	1	-	-	0	-	-
C	1	-	1	-	-	-	-	1	1	1	-	1	0	0	-	1
111 S	0	0	0	0	0	1	1	-	0	0	-	-	1	-	0	0
C	-	-	0	-	0	-	-	-	-	-	-	-	-	-	-	0

Table E.13. Fault table of the full-adder

Then, we deduce the fault tree, drawn in *Figure E.28*, allowing the diagnosis of the test sequence $\langle 000, 010, 111 \rangle$. To simplify the representation, all impossible situations are not represented. This tree partitions all the 33 possibilities (32 faults + one good state) into 11 groups. All the elements belonging to a group cannot be distinguished by the sequence (they are said to be equivalent with regard to this sequence). In particular, it is not possible to answer the question: "is the circuit faultless?".

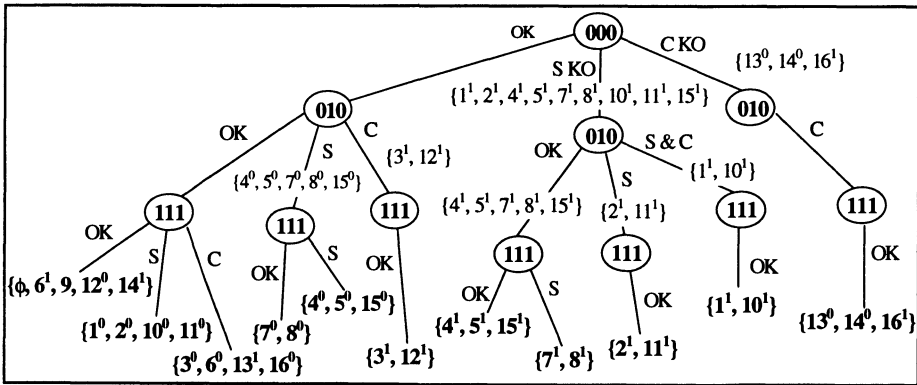


Figure E.28. Diagnosis tree of the full-adder

Exercise 13.11. Complete test sequence of a circuit

To be tested, each AND gate requires 4 input vectors: 111, 011, 101 and 110. As these two sets are not compatible, this leads to 8 different input vectors. Hence, all the 8 input vectors constitute the complete test sequence! As a consequence, the *exhaustive sequence* is also the optimal one.

Exercise 13.12. Redundancy analysis

1. Structural redundancy. A logical analysis gives: $f = a' + b.c$, $g = b.c + a.b.c$.

Function f has no structural redundancy; the gate 'abc' of g is structurally redundant. *Table E.14* gives the faults detected by all input vectors. It shows that 5 faults cannot be detected: 9^0 , 7^0 , 5^0 , 5^1 , 14^0 . They correspond to a passive structural redundancy.

abc	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
000	1	-	-	1	-	-	-	-	-	0	1	-	1	1	0	1
001	1	1	-	1	-	1	-	-	-	0	1	-	1	1	0	1
010	1	-	1	1	-	-	-	1	-	0	1	-	1	1	0	1
011	-	0	0	-	-	0	-	0	-	-	0	-	0	-	0	0
100	0	-	-	0	-	-	-	-	-	1	1	1	1	1	1	1
101	0	1	-	0	-	1	1	-	-	1	1	1	1	1	1	1
110	0	-	1	0	-	-	-	1	1	1	1	1	1	1	1	1
111	-	0	0	-	-	0	-	0	-	-	0	0	-	-	0	0

Table E.14. Fault table

2. Detection and diagnosis masking.

Detection masking. The fault noted γ in Figure E.29 cannot be detected. The fault noted α in the figure can be tested by the input vector (001). The output g takes '0' when this fault is present. But if γ is also present, the output g remains erroneously at value '1': hence fault α is no more detected.

Distinction masking. We assume that the untestable fault is present in the circuit. Its occurrence can lead to a wrong diagnosis if we apply the test sequence <110, 111> aimed at diagnosing between faults noted α and β . If the circuit is altered by fault α , this sequence will erroneously signal the presence of β because of a masking provoked by fault γ .

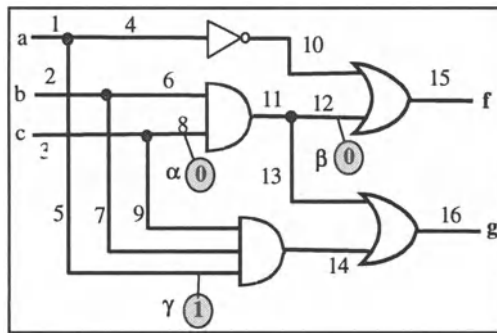


Figure E.29. Redundancy

Exercise 13.13. Structural testing of a program

1. The function `modify_temperature` increases or decreases the temperature, depending on the action parameter (1 = heating, 2 = cooling, else no action). The temperature is increased or decreased by a number of degrees function of the duration parameter value. The final temperature is then returned.

The main function `regulator` brings back the `initial_temperature` in the range 0 to 90 by a variation of 10 degrees if `variation = 0`, or a variation of 20 degrees else. It returns the `final_temperature` or -3000°C if the heater or the fan is damaged (`heating_state` or `fan_state = 0`).

2. We test the functional correctness of the regulator by analyzing the domains of the input parameters:
 - three cases for `initial_temperature < 0`, in `[0, 10]`, and `> 90`,
 - two cases for `heating_state = 0` or `1`,
 - two cases for `fan_state = 0` or `1`,
 - two cases for `variation = 0` or `1`.

As the domain of `initial_temperature` is not discrete, we test three values: `-50`, `30`, `150`, and the limits `0` and `90`. These values are combined with the discrete values of the other parameters.

3. When this sequence is applied, the coverage depends on the elements considered. For a *statement testing*, the coverage is not 100%. Indeed, the default part of the switch statement of the function `modify_temperature` is never executed.

Exercise 13.14. MC/DC testing of a program

Table E.15 gives a sequence of Boolean values for Condition/Decision, compatible with the requirements of MC/DC Testing.

Condition			Decision
A=B	C2	D>3	Action
True	True	True	True
False	True	True	False
True	True	True	True
True	False	True	False
True	True	True	True
True	True	False	False
True	True	True	True

Table E.15. MC/DC testing

Exercise 14.1. Ad Hoc techniques

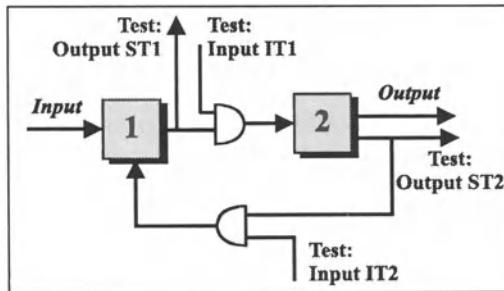


Figure E.30. Ad Hoc technique

We suppose that new inputs and outputs can be added to the circuit, in order to increase its controllability and observability. *Figure E.30* shows the modifications proposed to cut the feedback line between the two modules and to directly observe the outputs of each module.

Hence, two inputs (*IT1* and *IT2*) and two outputs (*ST1* and *ST2*) have been added. When *IT1* and *IT2* take the value '0', we block any uncontrolled evolution of the circuit. Hence, each module can be directly accessed.

Exercise 14.2. Analysis of a redundant circuit

1. We have: $f1 = a'c, f2 = a'c, f3 = a + c'$. The output variables do not depend on input *b*: thus, this circuit is redundant. In particular, the stuck-at 1 fault shown in *Figure E.31* cannot be detected from the inputs/outputs; indeed, to test this fault, we must satisfy:
 - the controllability constraint: $b = 0$,
 - the observability constraints: $a = 0$ and $c = 1$; this produces a '1' on line *x*, a '1' at the output of gate *A*, and finally a '1' at output *f1* which masks any detection of the fault.
2. The second circuit has a different logical behavior: $f1 = a' b' c + a b + b c'$, $f2 = a'c, f3 = a + c'$. The output *f1* is a logical function of input *b*. Obviously, this circuit could have been realized with a SIGMA-PI structure; however, it is a totally testable circuit.

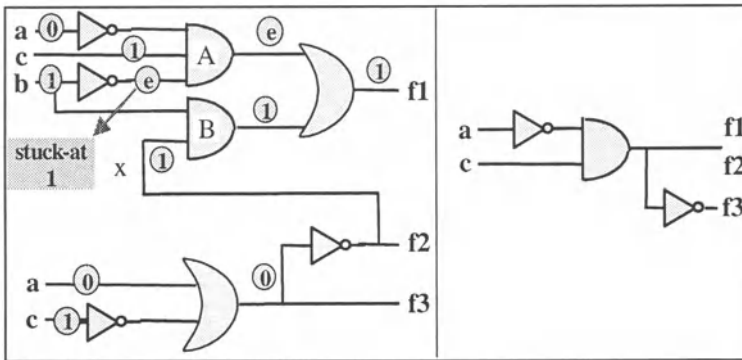


Figure E.31. Redundant circuit and its simplification

Exercise 14.3. Anti-glitch circuit

1. If gate *A* is removed from the given circuit, the logical function remains unchanged: $f = a b + b' c$. However, the output *f* can produce a glitch (a short negative pulse) when the inputs switch from (111) to (101). This anti-glitch circuit is useful but not completely testable: no stuck-at 0 of gate *A* are testable; indeed, their test requires $a = c = 1$, which implies that either the output of gate *B* or gate *C* is at '1', hence, the final output *f* always takes the value '1' (with or without fault). Consequently, this circuit has passive (untestable) redundancy, which cannot be removed!

- The anti-glitch circuit can easily be modified as shown in Figure E.32 in order to make it completely testable. When $T = 0$, the outputs of gates B and C are equal to '0', and we can test all stuck-at 0 faults of gate A.

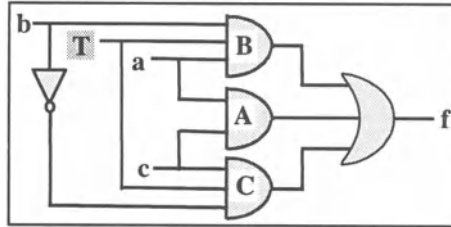


Figure E.32. Redundant circuit

Exercise 14.4. Easily testable gate network

The initial circuit requires 8 input vectors to be totally tested with the single stuck-at 0/1 fault model. It is an *exhaustive test sequence*.

In the modified circuit, input T must be at '0' during the normal functioning. This ensures that all XOR gates behave as INVERTERS, like in the initial circuit. During test operation, input T are set to '1', which applies the same inputs to the two AND gates. The circuit is completely tested with the 4-vector sequence of Table E.16.

a	b	c	T	comments
0	1	1	0	these three vectors detect all stuck-at '1' faults
0	1	0	1	
0	0	1	1	
1	1	1	1	detects all stuck-at '0'

Table E.16. Complete test sequence

Exercise 14.5. Reed-Muller structure

- Test sequence of the SIGMA-PI realization of the logical function:

$$TSO = \langle 1010, 1100, 0110, 1111, 1011, 1110, 0010, 0101, 1101 \rangle.$$

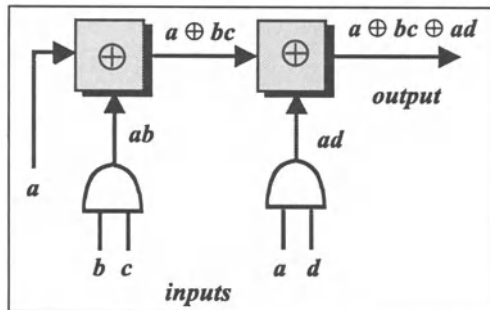


Figure E.33. RM circuit

2. We determine the logical expression of this circuit (see *Figure E.33*), and we compare this expression with the given SIGMA-PI expression. To facilitate this logical comparison, we may use an intermediate verification model such as a truth table (which corresponds to a canonical logical form).
3. A XOR network has a very interesting property concerning error detection: any single error occurring on one input is automatically transmitted to the output. Indeed, a single input error changes the parity of the number of '1' inputs. As a XOR network produces an output '1' if and only if an odd number of '1' is applied to it, any change in the input parity of '1' values provokes a modification of the output. The 4 vectors of sequence *TS1* apply to each AND gate the three testing configurations 11, 01 and 10. Hence, every fault of each AND gate is activated as an error which enters the XOR network, and is consequently propagated to the final output where it can be observed.
4. Electronics specialists have shown that a 2-input XOR gate is fully tested by the exhaustive input test sequence only. With the previous *TS1* sequence, this property is not satisfied. It is very easy to verify that the proposed 5-vector *TS2* sequence applies the 4 input configurations to any XOR gate.

Exercise 14.6. FIT PLA

1. $f1 = a'b + bc, f2 = ab' + bc$. Hence, 3 product terms are needed: $a'b, ab'$ and bc (common to both functions).
2. *Figure E.34* shows the symbolic structure of this PLA.

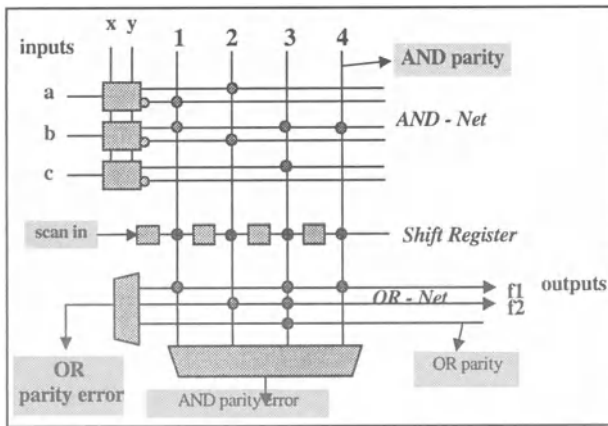


Figure E.34. FITPLA symbolic structure

3. Test sequence. It is made up of two parts: one sequence of 6 vectors to test the AND network, and a sequence of 4 vectors to test the OR network.

AND network test. $xy = 01 \rightarrow TS1 = \langle 011, 101, 110 \rangle$.
 $xy = 10 \rightarrow TS2 = \langle 100, 010, 001 \rangle$.

For example, the test vector 011 forces the first line of the OR matrix to take value '0', all other lines being at '1'. Thus, the 4 product terms take the values 1011;

hence, if no faults are present in the AND network, the parity error line is at '1'. Any fault equivalent to a stuck-at 0 of one active node (represented by a dot in the figure) belonging to columns 1, 3 and 4 is detected.

OR network test. A '1' bit is shifted 4 times from left to right in the shift register (scan in input). All single or multiple permanent hardware faults are detected, apart from the ones that do not modify the parity property of the AND parity vector (4 bits) and the OR parity vector.

Exercise 14.7. Scan Design

For each test vector V_i :

- The circuit is switched to *Test Mode*.
- A serial input operation through the *Scan In* input is performed, in order to load in the state register the 4-bit state belonging to the V_i test vector. This state loading operation takes 4 clock couples ($HM - HE$); in parallel, the state register containing the result from the previous test vector is read.
- The circuit returns to the Normal Mode, and one normal treatment step is executed with one clock pulse ($HM - HE$).

A last reading of the internal state register completes the test sequence.

Exercise 14.8. LFSR

1. This generator elaborates a deterministic cyclic sequence of 3-bit vectors. It is based on a synchronous shift register whose input is the XOR of bits 1 and 3. Hence, the initial condition gives the starting point of the sequence produced; this sequence is shown in *Table E.17*.

Clock	Q1 Q2 Q3
0	0 1 0
1	0 0 1
2	1 0 0
3	1 1 0
4	1 1 1
5	0 1 1
6	1 0 1

Table E.17. Sequence

2. The modified circuit behaves as a LFSR. From the initial state 111, we obtain the following cyclic output sequence: <111, 011, 001, 001, 100, 010, 110>. Let us note that the LFSR property is not guaranteed for any XOR feedback function. For example, if we take the XOR of bits $Q1$, $Q2$ and $Q3$, and if the initial state is 111, then the circuit remains always in this state!
3. We analyze in *Figure E.35* the evolution of the circuit from the initial state 100 when the first input vector 111 is received. Values in gray are the next state of the register. This study can easily be extended to the rest of the applied sequence.

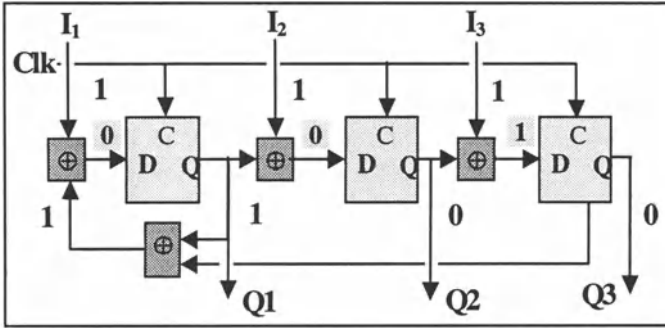


Figure E.35. Analysis of the PSA circuit

4. Non-detectable errors are necessarily multiple errors on several words of the incoming sequence. For example, is not detectable the modification of the first two words as follows (two single errors): <110, 111, ...>. Naturally, based on the mathematical properties of the Galois Fields, the class of non-detectable errors can be formally determined.
5. Such a BIST technique is very attractive, because the coding and decoding functions are easily implemented as logical circuits or software procedures. Moreover, the speed of the corresponding circuitry is high. Unfortunately, the efficiency of this technique in terms of fault detection strongly depends on the product to be tested.

FOURTH PART

Exercise 15.1. Single parity code

1. The redundant parity bit is obtained by making the XOR of all the other bits. The redundant codeword becomes (10111). This code detects:
 - any single error, i.e. 5 errors (the parity bit belongs to the codeword),
 - any triple error, i.e. 10 errors (all 3-out-of-5 words),
 - any quintuple error, i.e. 1 error (the word: 01000).

This makes 16 detected errors, amongst the $(2^5 - 1) = 31$ theoretically possible errors; hence, the error coverage of this simple code is $c = 16/31 = 0,516$.

2. Example of odd non-detectable error: 11101 (bits 2 and 4 are erroneous).
3. Characteristics of the code:

Capacity: $N = 2^{n-1} = 16$.

Density: $d = N / 2n = 16/32 = 0.5$.

Coverage rate for each codeword: $C = \text{number of detected errors} / \text{total number of possible errors} = 2^{n-1} / 2^n - 1 = 0.516$.

Redundancy: $rr = r / k = 1 / 4 = 0.25$ (or 1/5 of the codewords).

Exercise 15.2. Hamming Code C(7, 4)

1. This separable code adds to the initial bits three redundant bits, y_1 , y_2 and y_4 , calculated by the expressions given in the exercise text. One can easily deduce three properties, called *control relations*, which allow to detect and/or correct errors. We order these relations as follow to make error correction easier:

$$y_4 \oplus y_5 \oplus y_6 \oplus y_7 = 0 \quad (1), \quad y_2 \oplus y_3 \oplus y_6 \oplus y_7 = 0 \quad (2), \quad y_1 \oplus y_3 \oplus y_5 \oplus y_7 = 0 \quad (3)$$

We call *syndrome*, noted $s = (s_1, s_2, s_3)$, the vector obtained by computing these expressions. This syndrome is equal to zero if no error occurred; it is different from zero if a single error or a double error occurs. For example:

- if y_3 is false, expressions 2 and 3 are modified and the syndrome is $s = (0 \ 1 \ 1)$,
- if y_3 and y_6 are false, expression 1 is equal to '1', expression 2 remains at '0' (because two modifications are neutralized in a XOR function), and expression 3 is equal to '1'; hence $s = (1 \ 0 \ 1)$.

Erroneous bit		1	2	3	4	5	6	7
syndrome	s_1	0	0	0	1	1	1	1
	s_2	0	1	1	0	0	1	1
	s_3	1	0	1	0	1	0	1

Table E.18. Syndrome values

All multiple errors with rank higher than 2 cannot be detected. For example, if y_1 , y_2 and y_3 are false, the resulting syndrome is equal to '0', hence this triple error is not detected. If only single errors occur, they are detected. Moreover, a non-null decimal value of the syndrome indicates the position of the erroneous bit, as shown in Table E.18: this property justifies the chosen relation order.

2. Any 'double' error is confused with a 'single' error when considering the value taken by the syndrome. For example, we have shown that the double error altering y_3 and y_6 produces the syndrome value $s = (1 \ 0 \ 1)$: this error has the same effect than a single error altering y_5 .
3. This code is very close to the one presented in Example 15-4. Indeed, it corresponds to a simple re-organization of the coding relations. Consequently, both codes have the same detecting and correcting capability. The interest of the version of this exercise is only to facilitate the identification of the erroneous bit.
4. In order to allow the detection of single and double errors, and to allow the correction of single errors, we add a height redundant bit obtained by the \oplus of all the bits. This redundant bit add a fourth control relation:

$$y_1 \oplus y_2 \oplus y_3 \oplus y_4 \oplus y_5 \oplus y_6 \oplus y_7 \oplus y_8 = 0 \quad (4)$$

This relation produces the fourth bit of the syndrome, s_4 . Thanks to this fourth relation, we can distinguish between any single error, which lead to $s_4 = 1$, and any double error, which maintains $s_4 = 0$. This new code is called the modified *Hamming code C(8, 4)*.

Exercise 15.3. Linear code

1. Matrices G and H are deduced from the coding and control relations:

$$G = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix},$$

$$H = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}.$$

We verify that $H \cdot G^T = 0$:
$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

2. Coding: $Y = U \cdot G$, i.e. $[y_1, y_2, y_3, y_4, y_5, y_6, y_7] = [u_1, u_2, u_3, u_4] \cdot G$.

For example, if $U = [1 \ 1 \ 0 \ 1]$, then $Y = [1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1]$.

3. Detection and correction: $S = H \cdot W^T$, i.e. $H \cdot \begin{bmatrix} w1 \\ w2 \\ w3 \\ w4 \\ w5 \\ w6 \\ w7 \end{bmatrix} = \begin{bmatrix} s1 \\ s2 \\ s3 \end{bmatrix} = S$

If we analyze the codeword $W = [1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1]$, we can verify that $H \cdot W^T = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$.

If bit 3 is erroneous in this codeword, $H \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$ identifies the faulty bit.

Exercise 15.4. Encoding of a cyclic code

The first phase of the coding process uses 4 clock pulses and delivers at the output Y the higher bits of the codeword, i.e. the bits of the word to be coded u in the decreasing order: 1, 1, 0 and 0 (see *Figure E.36*).

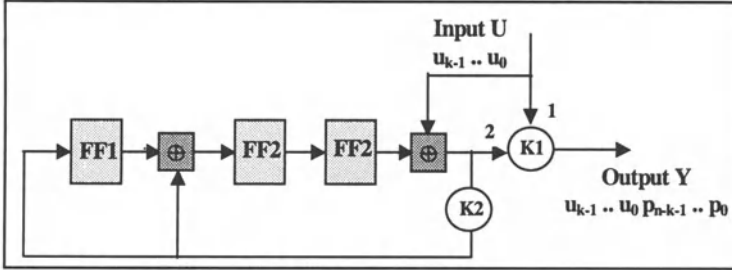


Figure E.36. Encoding circuit

Clock pulse	FF1	FF2	FF3
1	1	1	0
2	1	0	1
3	1	0	0
4	0	1	0

Table E.19. State evolution

During this phase, the state of the 3 D-Flip-Flops, initially at '0', evolves as shown in *Table E.19*. Then, the content of the register is shifted to the output Y . Hence, the codeword is $y = (0100011)$ corresponding to the polynomial: $y(x) = x + x^5 + x^6$. Now, let us calculate the codeword by a direct division of $x^{(n-k)} u(x)$ by $g(x)$, which gives $x^{(n-k)} u(x) + r(x)$:

$x^5 \cdot u(x) = x^6 + x^5$	$g(x) = x^3 + x + 1$
$x^6 + x^4 + x^3$	$x^3 + x^2 + x = \text{quotient}$
$0 + x^5 + x^4 + x^3$	
$x^5 + x^3 + x^2$	
$0 + x^4 + x^2$	
$x^4 + x^2 + x$	
$r(x) = x$	

We obtain the same codeword $y(x) = x^{(n-k)} u(x) + r(x) = x + x^5 + x^6$.

The generator matrix associated with this cyclic code is: $G = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$.

We verify that $[0\ 1\ 0\ 0\ 0\ 1\ 1] = [0\ 0\ 1\ 1] \cdot \begin{bmatrix} 1\ 1\ 0\ 1\ 0\ 0\ 0 \\ 0\ 1\ 1\ 0\ 1\ 0\ 0 \\ 1\ 1\ 1\ 0\ 0\ 1\ 0 \\ 1\ 0\ 1\ 0\ 0\ 0\ 1 \end{bmatrix}$.

Exercise 15.5. Single parity bidimensional code

This code uses redundancy at two levels: Longitudinal Redundancy Checking bits (noted LRC) are added to each word, and Vertical Redundancy Check words are added to the block (VRC). Each row and each column of the coded matrix belongs to an error detecting and correcting code.

	1	2	3	4	5
1	1	0	0	0	1
2	1	0	1	1	1
3	0	1	1	0	0
4	1	1	1	0	1
5	0	0	0	0	0

Table E.20. Bidimensional code with single parity

1. One parity bit is added to each word (LRC), and a word (VRC) is added to the block. Table E.20 gives an example of coding with $p = 5, k = 4$. After treatment (e.g. a memory storage), a parity check is applied to each word, and a parity check is made between all words. Any erroneous row or column is recorded.
2. Any single or multiple error is detectable if at least one error occurs on a row or a column. It is obviously the case for any odd multiple error. It is also the case for some even multiple errors; for example, a quadruple error on a same word will be detected four times.
3. To be undetectable, an error must have an even rank on each altered row and each altered column. For example, the quadruple error altering the bits of rows 1 and 3 and columns 2 and 4 cannot be detected, as no parity violation occurs.
4. Any error detected on rows 4 and 5 and columns 2 and 3 is a double error. Two errors can produce this signature (Table E.21), but we cannot identify which one is present:

	column 2	column 3		column 2	column 3
row 4	error			error	
row 5		error		error	

Table E.21. Errors detected on rows 4 and 5 and columns 2 and 3

5. Two classes of errors can be corrected:
 - All single errors. The erroneous bit is identified by intersection between the detected row and column; then, the identified bit must be complemented.

- All odd errors on a same row or a same column.

Note. The practical efficiency of such a code is strongly related to the technology used to store the data words. The error model considered here must be validated by a statistical fault analysis.

Exercise 15.6. *M-out-of-n* code

1. Let $w1$ and $w2$ any two different words of the *m-out-of-n* code. Having both exactly m bits '1', they are different for at least 2 bits. If we compute the OR function of these two words, the number of '1' bits will be at least $(m + 1)$ bits, and if we compute their AND function, the number of '1' bits will be at most $(m - 1)$. Thus, in both cases the resulting combined word does not belong to the *m-out-of-n* code. For example, if $w1 = 1100101$ and $w2 = 1001110$, then $w1$ OR $w2 = 1101111$, $w1$ AND $w2 = 1000100$, both outside the code.
2. The smallest distance between two words is 2, as seen in the previous question (e.g. 1100101 and 1101001). The greatest distance between two words is $2 \cdot m$ if $n \geq 2 \cdot m$, or $2 \cdot (n - m)$ if $n < 2 \cdot m$. Examples:
 - $d(1100101, 0011011) = 6$ for a 4-out-of-7 code,
 - $d(11001010, 00110101) = 8$ for a 4-out-of-8 code.
3. By definition, a unidirectional error modifies the number of bits '1' of the altered word; thus, this error is easily detectable by counting the number of bits '1'.
4. It is necessary to count the number of bits '1' of the word after treatment, and to compare this number with m . This operation can be performed by a specific logic circuit or a software procedure written in assembly language, according to the speed requirement of the final application: a circuit is more expensive but faster than a software procedure.

Exercise 15.7. Berger code

1. If $k = 4$, we need $r = 3$ redundant bits to express the number of '0' contained in the data part. Thus, this code is not optimal (with $r = 3$ we could have $k = 7$). Table E.22 shows all the obtained codewords.

X a b c d	R e f g	X a b c d	R e f g
0 0 0 0	1 0 0	1 0 0 0	0 1 1
0 0 0 1	0 1 1	1 0 0 1	0 1 0
0 0 1 0	0 1 1	1 0 1 0	0 1 0
0 0 1 1	0 1 0	1 0 1 1	0 0 1
0 1 0 0	0 1 1	1 1 0 0	0 1 0
0 1 0 1	0 1 0	1 1 0 1	0 0 1
0 1 1 0	0 1 0	1 1 1 0	0 0 1
0 1 1 1	0 0 1	1 1 1 1	0 0 0

Table E.22. Berger code for $k = 4$

2. The *Berger* code is separable and can thus be structured into two fields (X, R), where X is the word before coding and R the redundant field. R is the binary number of '0' bits of X . Let us first consider a unidirectional fault that increases the number of '0' bits of the complete word. Three cases can be considered:
 - If the X field only is altered, the number of '0' of X is increased and becomes greater than the value in R : hence, this error is detected, as $\text{NbZero}(X) > R$,
 - If the R field only is altered, the value of R decreases whilst the real number of '0' bits of X is not modified: here also this error is detected, as $R < \text{NbZero}(X)$,
 - If X and R are both altered, the value of R becomes again smaller than the number of '0' of X which has increased, so the error is detected.

The reasoning is similar with a unidirectional error that reduces the number of '0' bits of the complete word.
3. Now, R is the binary expression of the number of '1' in X . We follow the same reasoning as in the previous question with first an error that increases the number of '0' bits of the complete word:
 - If the X field only is altered, the number of '1' of X is decreased and becomes smaller than the value in R : hence, this error is detected, as $\text{NbOne}(X) < R$,
 - If the R field only is altered, the value of R decreases whilst the real number of '1' bits of X remains unchanged: here also this error is detected, as $R < \text{NbOne}(X)$,
 - If X and R are both altered, the value of R decreases whilst the number of '1' bits of X also decreases: the error is not necessarily detected.

Exercise 15.8. Unidirectional codes

The comparison between the capabilities of these codes is presented in Appendix A.

For $n = 10$: the 5-out-of-10 code gives 252 codewords, the double-rail 5/10 code has 32 codewords, and the ($m = 7, r = 3$) code has 128 codewords.

Exercise 15.9. Modulo 9 proof

1. The class of any integer A modulo 9 is the remainder of the division of A by 9.

If $A = A_0 + A_1 10^1 + A_2 10^2 + \dots + A_n 10^n$, its class, noted $C(A)$, is:

$$C(A) = (A_0 + A_1 10^1 + A_2 10^2 + \dots + A_n 10^n) / 9 \text{ [MOD 9]},$$

$$C(A) = A_0 + A_1 + \dots + A_n \text{ [MOD 9]}, \text{ as any power of 10 gives 1 as remainder.}$$

This means that the research of the remainder of a division of an integer by 9 is equivalent to the determination of the remainder of the sum of the figures of this integer by 9. And this process is iterative. For example, if $A = 591$:

$$C(A) = 5 + 9 + 1 \text{ [MOD 9]} = 15 \text{ [MOD 9]} = 1 + 5 \text{ [MOD 9]} = 6 \text{ [MOD 9]}.$$

This procedure is very simple to implement as hardware or software module.

2. Verification of the operation:

$$189 = 1 + 8 + 9 \text{ [9]} = 18 \text{ [9]} = 9 \text{ [9]} = \mathbf{0} \text{ [9]}, \quad 47 = 11 \text{ [9]} = \mathbf{2} \text{ [9]}.$$

- We perform the addition of the classes of the two considered numbers,

$0 + 2 = 2$ [9], and we observe that the resulting class belongs to the same class as the expected final result: $236 = 2 + 3 + 6$ [9] = 11 [9] = 2 [9].

- In order to verify the second operation, we multiply the two classes, $0 \times 2 = 0$ [9], value which is different from the class of the expected final result: $8867 = 8 + 8 + 6 + 7$ [9] = 29 [9] = 11 [9] = 2 [9].
- The third operation is verified by subtracting the two classes of the numbers, $0 - 2 = 7$ [9], value which is different from the expected result: $144 = 1 + 4 + 4$ [9] = 9 [9] = 0 [9].

Hence, we have detected an error on the operations 2 and 3. However, we cannot correct those errors, as this code is only an error detecting code. Moreover, all the faults are not detected, as shown by the fourth operation:

- $189 - 47$ [9] = 7 , and 97 [9] = 7 ; however, the correct value is $198 - 47 = 142$, which is different from the proposed result 97 . This conclusion is generalized by question 4.
3. With the example $48 / 12 = 4$, we obtain $48 = 3$ [9], $12 = 3$ [9], $3 / 3 = 1$, value which is different from the class of the correct result: 4 .
 4. An error transforms a result N into another number $N^* = N + E$ (E is the error, either positive or negative). This error is not detectable if and only if $N^* = N$ [9], that is to say, if E is a multiple value of 9.

Exercise 15.10. Binary residual code

1. The binary number can be expressed as: $N = N_0 16^0 + N_1 16^1 + \dots$. The remainder in the division by $15 = 2^4 - 1$, is obtained as in the previous exercise, by an iterative process on the N_i elements. Practically speaking, we divide the binary numbers into 4-bit slices (as $16 = 2^4$), starting from the LSB (the least significant bit), and going towards the MSB (most significant bit). If necessary, some '0' value bits can be added to the left of the number to complete the last slice. Then, these slices are added together modulo $15 = 1111$.

- $N = 0010\ 1111\ 1011\ 1100\ 1101$
(two '0' bits have been added to the left of the number),
- $N = 0010 + 1111 + 1011 + 1100 + 1101$ [15] = $11\ 0100$ [15] = $0011 + 0101$ [15]
= 1000 [15]
(this result can be obtained directly or, on the contrary, by progressively adding the slices two by two).

2. Verification of the operation:

$0011\ 0010 + 0110\ 1110 = 1010\ 1100$? In decimal, this gives: $50 + 110 = 172$?

Classes of the two operands and of the expected result: 0101 (5), 0101 (5), 0111 (7).

We add the two classes of the operands: $0101 + 0101 = 1010$ (10).

We do not find the class of the expected result. Hence, the operation is false (this can easily be manually verified in decimal).

Exercise 15.11. Checksum code

1. These five words are added without carry. The resulting word is joined to the

others, hence constituting a block of six 4-bit words. Here also, this computation can be made, either globally, or in a cumulative way:

- $1101 + 0011 = 0000$ (the carry is ignored),
- $0000 + 1110 = 1110$,
- $1110 + 0110 = 0100$,
- $0100 + 0101 = 1001$ which is then complemented to '2': $1\ 0000 - 1001 = 0111$.

This last word is then added to the block.

2. The stored block contains 6 words: (1101, 0011, 1110, 0110, 0101, 0111); indeed, the addition of the 5 first words gives the previous 1001 value which is by construction the 2's complement of the fifth word 0111. Their addition modulo 2^4 gives 0000.
- 3 - 4. This modulo 2^4 code detects any error that does not add or subtract to the correct result a value which is a multiple of 16.

Exercise 15.12. GCR(4B - 5B) code

Obviously, this mapping is redundant: 5 bits are needed instead of 4. Its property is to ensure that at most two successive zeros occur:

- in a word, as there are no more than 2 successive '0' bits,
- in consecutive words in a serial transmission, as data combinations with more than one zero at the beginning and the end of any word are prohibited.

This property is desirable in some applications (transmission, storage) to increase bit density (e.g. for data storage on a magnetic tape) and ease clock synchronization.

Exercise 16.1. Test of a control system

1. The test of regulators $R1$ and $R2$ is performed off-line, according to a cyclic mode. A real-time clock periodically activates the test task. An efficient testing procedure will require a proper access to the regulators in order to test their various regulation functions and process interfaces: amplifiers, sample and hold modules, analog to digital and digital to analog converters, etc. The tester module can, for example, order the regulators to perform some pre-defined regulation treatments, then to compare the obtained results with correct values stored in memory. The periodicity of the test is here of 168H for $R1$ and 24H for $R2$. If we know the reliability of the equipment, we can deduce the probability of the occurrence of a fault between two consecutive tests: assuming simple exponential laws with constant failure rates, the fault probability during the test is approximated to the value: $test\text{-}period \times \lambda$.

Dealing with $R3$, an interrupt procedure is envisaged with a time slot of 10' every hour: hence, the test period is 1H. We suppose that during this 10' test operation, $R3$ is totally checked.

- 2 - 3. On the contrary, if the complete test of the regulator is longer than 10', it is necessary to split the testing task into several shorter test sequences, for example one 8' and one 7' sequences. Thus, the periodicity of the complete test is increased to 2 hours.

Exercise 16.2. Duplex technique

1. Any multiple fault altering the functional module is detected as soon as it produces an output error (failure of this module). This also stands for any fault altering the duplicate module.

Faults altering the comparison module are detected only if they lead to an incorrect error signal value. For example, the stuck-at 0 of this erroneous output cannot be detected if '0' is considered as the specification of a correct result.

The undetectable faults in the functional modules are those that modify in the same way and at the same time both duplicated modules: they provoke the same failure. This is the reason why these duplicate modules must be realized with different methods and technologies.

Note. When an error is detected, it is not possible to locate it.

2. The number of faults of this product is about twice the number of faults of the basic module; hence, the fault probability of the product is twice the fault probability of one module. Consequently, the reliability of a duplex is lower than the reliability of a non-redundant product. This is the price to pay for an immediate detection (on-line testing technique) of the failures.

Exercise 16.3. On-line testing of a half-adder

1. *Table E.23* provides again the truth table of the half-adder. We observe that, whatever its structural implementation, this half-adder possesses natural functional redundancy: the output vector $(s\ c) = (1\ 1)$ never occurs. An external observer can exploit this property in order to detect 'on-line' any fault producing a failure characterized by this forbidden vector (a simple AND gate is sufficient to detect this case). However, this on-line testing capacity is very limited and covers only a few real faults; in particular no stuck-at 0 fault can be detected on-line.

a b	s c p
0 0	0 0 0
0 1	1 0 1
1 0	1 0 1
1 1	0 1 1

Table E.23. Truth table

2. *Figure E.37* shows the modified gate circuit and the corresponding truth table when a parity output p is added to this half-adder. Error detection is performed by a 3-input XOR gate.

In that case, a separate structural redundant circuit has been added to the basic circuit. The on-line testing capability is better than in the previous technique. However, some faults are still not tested, such as the stuck-at '0' noted α on the figure: indeed, if $a = 1$ and $b = 1$, this fault produces the undetected failure $(s\ c\ p) = (1\ 0\ 1)$ instead of the normal vector $(0\ 1\ 1)$.

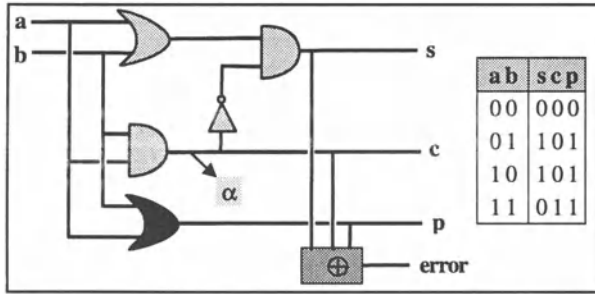


Figure E.37. Half-adder with a parity output

3. In order to improve this situation, the previous circuit is modified by using three independent circuits (Figure E.38). Any fault altering only one of these three independent circuits is detected as soon as it provokes an error at one output only. Hence, this on-line detection capability concerns all faults belonging to the stuck-at fault model. However, the detection circuit is not concerned by the on-line testing property. Indeed, the stuck-at 0 of the output of this circuit is not detected! To remedy this problem, we can use a self-checking circuit, as shown in the right part of Figure E.38. The final error outputs *f* and *g* belong to the 1-out-of-2 detecting code {10, 01}. Hence, any single fault in the whole circuit is now detected.

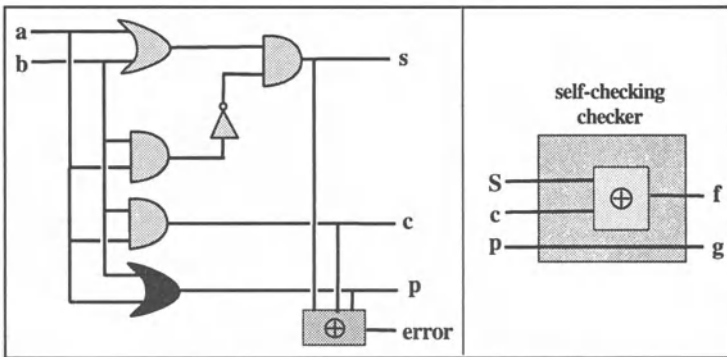


Figure E.38. Use of independent circuits and corresponding SCC

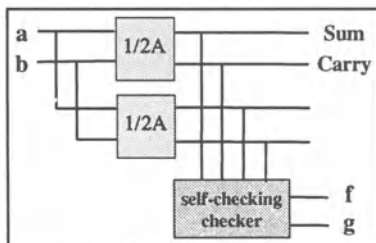


Figure E.39. Duplex approach

4. A Duplex structure is shown in *Figure E.39*. It uses two half-adder modules and a 2-bit double-rail SCC (this SCC is studied in the next exercise). We suppose that the two duplicated modules are not affected by the same faults simultaneously. The advantage of such approach is its simplicity. On the contrary, it is much more expensive in terms of gate number.

Exercise 16.4. Double-rail self-checking checker

1. If each input pair (a_1, a_2) and (b_1, b_2) belongs to the set $\{01, 10\}$, 4 input vectors (2^2) can be applied to this SCC during a normal operation of the tested circuit. This circuit (see *Figure E.40*) uses 4 product terms: $A = a_1 \cdot b_2$, $B = a_2 \cdot b_1$, $C = a_1 \cdot b_1$, and $D = a_2 \cdot b_2$.

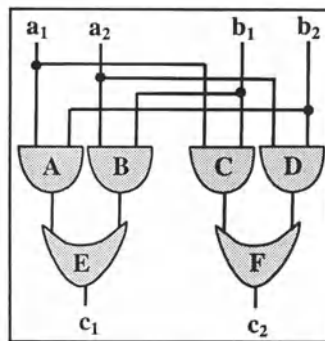


Figure E.40. Double-Rail SCC

It is easy to verify on the circuit that, in each case, only one AND gate (A , B , C , and D) is active, hence producing an output vector belonging to the set $\{01, 10\}$. All other input vectors (12 vectors) produce different output values:

- if the number of inputs '1' is lower or equal to 2, no AND gate is active and the output vector is 00,
- if the number of inputs '1' is greater than 2, at least one AND gate of each output is active, producing an output vector 11.

This behavior is shown by *Table E.24*. We can deduce from this table that this circuit is *code disjoint*.

Now, to prove that this circuit is a SCC for the 2-bit double-rail code, we must prove that it is *self-testing* for the normal input vectors. So, all its stuck-at faults must be tested during the normal operation, i.e. by application of the previous four codewords only! Obviously, the circuit presents symmetry property between the AND and the OR gates. We see in *Table E.24* that each AND gate is activated once and is activated alone; hence, all stuck-at 0 are tested by producing an output vector 00 which is outside the code $\{01, 10\}$. Let us consider the gate A ; it receives the input vectors 01 (input codeword 0101) and 10 (input codeword 1010), and each time gate B is inactive: hence, all stuck-at 1 of gates A and E are tested by producing an output vector 11 outside the normal code. Symmetrical situations can be found for all other AND gates.

Inputs	a1 a2 b1 b2	c1 c2
2 / 4 codewords (4 vectors)	0 1 0 1 D	0 1
	0 1 1 0 B	1 0
	1 0 0 1 A	1 0
	1 0 1 0 C	0 1
wrong 2 / 4 words (2 vectors)	1 1 0 0	0 0
	0 0 1 1	0 0
less than 2 bits '1' (5 vectors)	0 0 0 0	0 0
	----	--
	1 0 0 0	0 0
more than 2 bits '1' (5 vectors)	0 1 1 1	1 1
	---	--
	1 1 1 1	1 1

Table E.24. Truth table of the SCC

2. Let us analyze the global circuit of Figure E.41 which combines 3 elementary checkers to check a 4-bit double-rail code. To prove that this circuit is a SCC, it is sufficient to verify that each checker receives the four 2-bit double-rail codewords defined in question 1.

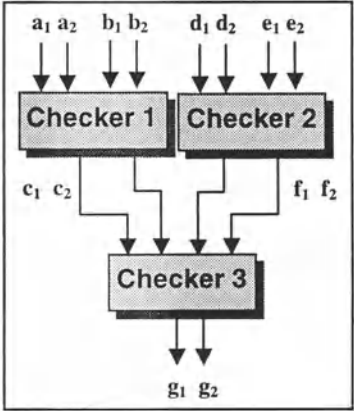


Figure E.41. Association of three SCCs

Test vectors	Internal	Outputs
a b d e	c f	g
01 01 01 01	01 01	01
01 10 01 10	10 10	01
10 01 10 10	10 01	10
10 10 10 01	01 10	10

Table E.25. Minimum test sequence

Table E.25 shows that the whole SCC is tested by a sub-set of only 4 input codewords: each checker receives a testing set of 4 input vectors.

Exercise 16.5. Parity self-checking checker

1. The circuit (Figure E.42) is a SCC converting a 4-bit odd-parity input code into a 1-out-of-2 output code. We must verify that it is code disjoint and self-testing.

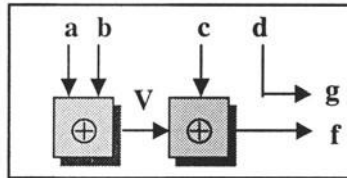


Figure E.42. Parity checker

- The circuit is obviously ‘code disjoint’. If an odd number of inputs (1 or 3) take the value ‘1’ (8 cases), the outputs f and g belong to the code {01, 10}. On the contrary, if an even number of inputs (0 or 2) take the value ‘1’, the output signals take the values 00 or 11.
- We assume that the test of each XOR gate requires the application of all its input vectors (00, 01, 10 and 11). Any error is then propagated to the final output, as XOR gates propagate any input modification (the observability of a XOR network is complete for single errors).

Table E.26 shows an example of test sequence constituted of four input vectors belonging to the normal odd-parity input code. This 4-length sequence is minimal. So, the circuit is self-testing. It is a self-checking checker for an odd-parity code.

Lines					Outputs	
a	b	V	c	d	f	g
0	0	0	0	1	0	1
0	1	1	0	0	1	0
1	0	1	1	1	0	1
1	1	0	1	0	1	0

Table E.26. Minimal test sequence

2. We know from question 1 that a subset of only 4 input codewords is sufficient to ensure the self-testing property. However, not any subset guarantees this property: for example, the circuit is no longer self-testing if the circuit under on-line testing produces the first three codewords only.
3. Consider the minimal test sequence given in question 1. If we operate a permutation of inputs b and c , the resulting set of input vectors does not provide the self-testing property to the circuit. Indeed, the second XOR gate receives two input vectors only (00 and 01).

Exercise 16.6. Software functional redundancy

First of all, the used formal parameters and local variables represent temperatures lower than 0°C, as the function is called only if the freezer is freezing. In the present case, the function calling with $Min = +372$ °C, or the return of a positive I value, will not be detected. To remedy this problem, we introduce a new type `Freezing_Temperature`:

```
subtype Freezing_Temperature is integer range
    Minimal_Temperature .. 0;
```

where `Minimal_Temperature` is an negative constant previously declared. Hence, Min , Max and I belong to this type.

Moreover, this function implicitly assumes that the value of Min should be lower than the value of Max . However, no verification of this property is made. We propose to add to the program a pre-condition as the first statement of the body of the function:

```
if Min > Max then raise Erroneous_Call;
end if;
```

Finally, the returned value must belong to the range $[Min, Max]$. Here also, this condition is implicitly expressed by the name of the function. However, the violation of this property due to a design fault is not detected. We propose to add just before the 'return' statement the post-condition:

```
if not (Min<=I and I<=Max) then raise Erroneous_Design;
end if;
```

Exercise 17.1. Traffic Light Controller

Figure E.43 gives the coded state table and the symbolic Moore structure of the traffic light controller.

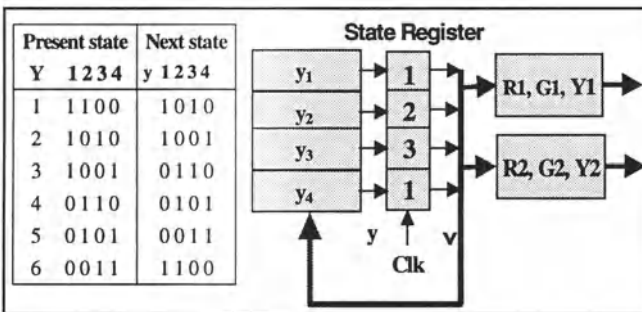


Figure E.43. Fail-Safe design of the controller

1. The 2-out-of-4 code can represent $N = \binom{4}{2} = 6$ codewords, which is exactly the number of internal states of the state graph to be coded.
2. Four synchronous D Flip-Flops are used to implement this circuit. The D-inputs ($Di = yi$) are logical functions of the outputs of these Flip-Flops ($Qi = Yi$):

$$D1 = Q1.Q2 + Q1.Q3 + Q3.Q4$$

$$D2 = Q1.Q4 + Q2.Q3 + Q3.Q4$$

$$D3 = Q1.Q2 + Q1.Q4 + Q2.Q4$$

$$D4 = Q1.Q3 + Q2.Q3 + Q2.Q4$$

They are realized by 4 independent AND/OR logical networks. The outputs are also realized by monotonic circuits (AND/OR gates only) of the outputs of the D flip-flops:

$$R1 = Q1.Q2 + Q1.Q3 + Q1.Q4 + Q3.Q4, \quad Y1 = Q2.Q4, \quad G1 = Q2.Q3$$

$$R2 = Q1.Q4 + Q2.Q3 + Q2.Q4 + Q3.Q4, \quad Y2 = Q1.Q3, \quad G2 = Q1.Q2$$

3. We will examine a few faults of the D_i equations to show the principle of the on-line detection. Any stuck-at 0 altering an AND gate will be transformed into an error ($1 \rightarrow 0$) when the present state of the FSM normally activates this gate; hence, the next state will have one bit '1' only. Consequently, all the AND gates will take an output value '0', and, at the next clock pulse, the FSM will reach a null state (0000). The convergence towards this null safe state is achieved in two clock pulses. Moreover, this is a stable trap state. A same reasoning applies to any stuck-at 0 fault of the OR gates. Now, let us consider a stuck-at 1 fault in a AND gate which provokes an error ($0 \rightarrow 1$), for example gate $Q1.Q2$ of $D1$. A necessary condition for this error to be propagated to $D1$, is that the present state is different from (1100). Whatever the considered stuck-at 1 fault, there is a present state that leads the FSM to a state having 3 bits '1'. The structure of the D_i expression is such that, in that case, the next state will be the stable trap safe state (1111). Here also, the convergence is made in two clock pulses.

To complete this study, we could ask the question: is any stuck-at fault detected during the normal functioning of this FSM? The answer is 'yes', if we assume that the whole state graph is totally used (each state and each transition) during the normal life of the circuit. This is required to guarantee the implicit single fault assumption. If not, we could have double faults that inhibit the fail-safe property.

4. Any fault affecting one flip-flop will provoke an evolution of the internal state of the circuit outside the 2 / 4 code, like in the previous question. If the Clock input is blocked (stuck-at 0 or 1), the whole state machine remains in the same correct state; hence, this fault is not safe. To remedy this problem, the specialists have proposed special duplicated clock systems.
5. With a 1-out-of- n coding of the internal states, we need 6 internal variables instead of 4. However, the logical expressions are very simple.

Exercise 17.2. Mathematical function processing

According to the first approach, if the treatment is stopped, no value is available for Y . On the contrary, after each iteration of the second approach, a value of Y is available and this value is closer and closer to the correct result. Consequently, an approximate value may be used if the deadline is reached before the end of the normal processing.

So, this second solution is much preferable to implement a fail-safe program.

Exercise 18.1. Reliability of the TMR

1. There is no failure as long as 2 of the 3 modules function correctly. We suppose that the three modules have the same reliability, $R_0(t) = e^{-\lambda t}$, and that the voter is faultless. The global reliability corresponds to all situations leading to no failures. We can determine it by different methods.

- We enumerate all these statistical situations: 2 modules are faultless and the third one is faulty (3 cases), and the three modules are faultless (1 case).

Thus, we obtain: $R = 3 \cdot R_0^2 \cdot (1 - R_0) + R_0^3 = 3 \cdot R_0^2 - 2 \cdot R_0^3$.

- We make a logical treatment based on the theorems of composed probabilities:

$$R = P(AB \text{ OR } AC \text{ OR } BC) = P(AB \text{ OR } (AC \text{ OR } BC)) = P(AB) + P(C \cdot (A \text{ OR } B)) - P(ABC),$$

P being the probability and A, B and C being the 3 modules.

Let us note that $P(ABC)$ is subtracted as it is the only event counted twice in the two other terms.

$$R = P(AB) + P(C) \cdot P(A \text{ OR } B) - P(ABC),$$

$$R = P(A) \cdot P(B) + P(C) \cdot (P(A) + P(B) - P(A) \cdot P(B)) - P(A) \cdot P(B) \cdot P(C),$$

$$R = P(A) \cdot P(B) + P(C) \cdot (P(A) + P(C) \cdot P(B) - 2 \cdot P(A) \cdot P(B) \cdot P(C)),$$

$$\rightarrow R(t) = 3 \cdot R_0(t)^2 - 2 \cdot R_0(t)^3 = 3 \cdot e^{-2\lambda t} - 2 \cdot e^{-3\lambda t},$$

as all modules have the same reliability.

By mathematical integration of the previous function, we deduce the MTBF (or the MTTF): $MTBF = 5/6 \text{ MTBF}_0$, which is lower than the MTBF of one module.

Note. In fact, the reliability curve of the TMR has a horizontal asymptote for $t = 0$; this reliability is greater than the reliability of the basic module when t is ‘small’, but it becomes lower after a certain time (see Appendix B).

2. According to the reliability diagram, the voter module is in ‘series’ with the three modules. Thus, its reliability must be multiplied by the reliability of the triplex:

$$R(t) = (3 \cdot e^{-2\lambda t} - 2 \cdot e^{-3\lambda t}) \cdot e^{-\lambda t/10}$$

Exercise 18-2. Fault tolerance of the TMR

In terms of reliability, we assume that the *TMR* system fails as soon as two modules fail. To simplify, we neglect the reliability of the Voter. Any functional or technological fault altering only one module is tolerated. In fact, such a redundant structure tolerates much more faults: any fault altering one or several modules is tolerated if and only if it does not modify the behavior of 2 or 3 modules in the same way and at the same time. For example, a fault producing the same simultaneous error at two module outputs induces a global failure of the system. Moreover, the latency phenomena slightly complicate this analysis. Indeed, we know that a fault is not necessarily activated as a failure as soon as it occurs. Thus, the tolerance is increased as the occurrence of a possible failure on 2 or 3 modules is delayed.

Let us now examine the *TRM* structure with hardware fault hypotheses. In an electronic circuit made of a set of components, faults are supposed to be independent probabilistic events (and we use probability theorems with this assumption). The assumption of a fault altering one module only, generally used (see the reliability computation made in the previous exercise), is justified by the fact that the

probability of having a double fault affecting two modules is the product of the probabilities of having a fault in each module. With electronic components, the actual values of the λ are very small (e.g. 10^{-7}), hence, we neglect the product terms (10^{-14}). This assumption cannot be made if strictly identical components are used in the *TMR*. Indeed, these components can have the same design faults or environmental weaknesses (e.g. sensitivity to temperature); thus, faults cannot be considered as independent phenomena and all reliability computations are false. Another criticism deals with other faults violating the independence assumption. They produce failures at the same time on non-identical components. For instance, this situation can result from external perturbations, such as an Electro-Magnetic parasite.

Exercise 18.3. *NMR*

- Let us assume that each module has only one output. During a normal functioning, the output vector $(z1, z2, z3)$ must take the values $(0\ 0\ 0)$ or $(1\ 1\ 1)$. Any other value is erroneous, hence the detection function is:

$$\text{error} = (z1' \cdot z2' \cdot z3' + z1 \cdot z2 \cdot z3)'$$

where '+', '.', and "'" represent the operators OR, AND, and NOT.

This expression can directly be implemented by a very simple circuit (containing few MOS transistors). We will develop it further to make a transition with question 2. We obtain the expression given in Chapter 18:

$$\text{error} = (z1 \oplus z2) + (z1 \oplus z3) + (z2 \oplus z3).$$

The 2-input \oplus operation gives a '1' if and only if its inputs are different.

- First we create the three elementary comparison functions:

$$fa = (z1 \oplus z2), fb = (z1 \oplus z3), \text{ and } fc = (z2 \oplus z3).$$

If $z1$ is erroneous, fa AND fb is equal to '1',

If $z2$ is erroneous, fa AND fc is equal to '1',

If $z3$ is erroneous, fb AND fc is equal to '1'.

Hence: $M1 = fa$ AND fb , $M2 = fa$ AND fc , $M3 = fb$ AND fc .

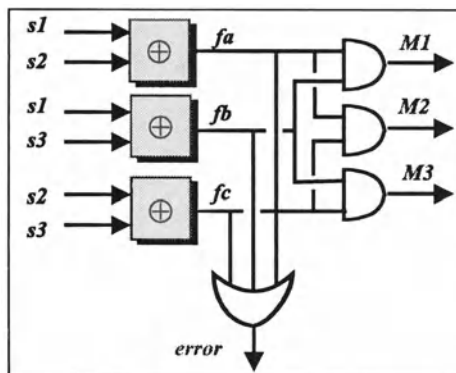


Figure E.44. Detection/Diagnosis circuit

The corresponding circuit is shown in *Figure E.44*. The signals *M1*, *M2* and *M3* identify the failing module (their value is a 1-out-of-3 codeword in case of error), and allow its inhibition (thanks to a power switch-off, for example), and finally its replacement by a spare module.

3. The voter must behave as the majority of its inputs: this function is the logic MAJORITY. For 3 inputs, we have:

$$\text{MAJ}(z1, z2, z3) = z1.z2 + z1.z3 + z2.z3.$$

The corresponding electronic CMOS component is simple.

This function can easily be extended to 4 inputs:

$$\text{MAJ}(z1, z2, z3, z4) = z1.z2.z3 + z1.z2.z4 + z1.z3.z4 + z2.z3.z4.$$

Note: The MAJORITY function is not associative (no possibility for combining smaller MAJORITY modules).

Exercise 18.4. Study of the double duplex

1. Reread Chapter 18, sub-section 7.2.2.
2. The product functions correctly as long as one of the two couples (1.1, 1.2) or (2.1, 2.2) functions correctly. The reliability of the product is then:

$$R = P((1.1 \text{ AND } 1.2) \text{ OR } (2.1 \text{ AND } 2.2)) = P(1.1 \text{ AND } 1.2) + P(2.1 \text{ AND } 2.2) - P(1.1 \text{ AND } 1.2) . P(2.1 \text{ AND } 2.2),$$

$$R = P(1.1) . P(1.2) + P(2.1) . P(2.2) - P(1.1) . P(1.2) . P(2.1) . P(2.2),$$

where + and - are the addition and subtraction operators.

If the modules have the same reliability R_0 , we have:

$$R = 2 . R_0^2 - R_0^4 = 2 e^{-2\lambda t} - e^{-4\lambda t}.$$

Note. The reliability curves of this structure are given in Appendix B.

Exercise 18.5. Study of self-purging technique

1. The switch-off of a failing module is performed by each one of the modules of the structure. This approach is interesting because it eliminates a part of the centralized commutation unit which is always a delicate part of a fault-tolerant system (such a part is called the *kernel* of the system). This technique is a step towards a complete decentralization of the duplicate modules and of the decision function (thanks to a distributed voter). Such distributed structure can be encountered in the framework of distributed software tasks in a distributed multiprocessor system.
2. When only 2 modules remain active, the product regresses to a simple *Duplex*. Thus, the next error occurrence will not be tolerated. The system operates according to a degraded mode until a maintenance operation restores the tolerance capacity of the product.

Exercise 18.6. Example of a tolerant program based on retry mode

The fault to be treated being associated with a provided data, the use of the *retry mode* is pertinent. Indeed, the Get procedure is not the cause of the problem. Fault

tolerance mechanism must detect an error if a non-integer data is provided from the keyboard. In this case, the data sampling must be reiterated (the `Get` function is executed again) after having restored the initial context. Let us consider the following solution:

```

Procedure Safe_Get(I : out integer) is
begin
  loop
    begin
      Get(I);
      exit;
    exception when Data_Error => Skip_Line;
    end;
  end loop;
end Safe_Get;

```

As soon as an integer value is provided and acquired by the `Get(I)` function, the `exit` statement allows to exit from the loop (`loop`), hence to finish the execution of the `Safe_Get` procedure.

On the contrary, the reading by `Get` of an erroneous data value leads to the raising of an exception (`Data_Error`) and the branching to the associated exception handler. This treatment erases (thanks to the operation `Skip_Line`) the content of the buffer containing the keypressed characters; these characters have not yet been all extracted because of the partial execution of `Get`. For example, if the user has keypressed the 5-character sequence `<17A28>`, and then 'Carriage Return', the execution of `Get(I)` can let characters '2' and '8' in the keyboard buffer, as the left to right analysis of the expected figures has been interrupted by the raising of the exception induced by the analysis of 'A'. As expected, the buffer reset action restores the system in a safe state.

Exercise 18.7. Programming and evaluation of recovery bocks

1. Programming. The two following program extracts illustrate the two approaches proposed in section 18-4. We assume that the execution context is limited to the input/output parameter `C`. `C_Prime` is a data structure having the same type `T` as `C`. It will store the safeguard copy (the duplicate).

```

Procedure Recovery_Block_V1(C : in out T) is
  C_Prime: T;
  Error: Boolean;
begin
  Save(C, C_Prime);
  Error := P(C);
  if Error then Restore(C_Prime, C);
    Q(C);
  end if;
end Recovery_Block_V1;

```

Where the procedures Save(X, Y) and Restore(X, Y) both makes a copy of X into Y.

```

Procedure Recovery_Block_V2(C : in out T) is
  C_Prime : T;
  Error: Boolean;
begin
  Save(C, C_Prime);
  Error := P( C_Prime );
  if Error then Q( C );
                else Restore(C_Prime, C);
  end if;
end Recovery_Block_V2;

```

2. Evaluation of the performance. The two previous programs allow the expected performance of the two proposed approaches to implement the recovery blocks to be evaluated. In both cases, the context is initially saved. But the rest of the bodies is different.

- In the first case, a correct execution of P does not require any supplementary treatment; when an error is detected, a restore operation is performed before executing procedure Q.
- In the second case, an opposite situation occurs, i.e. a correct execution implies a restore operation; on the contrary, in case of error detection such restore operation is not necessary before executing Q.

To conclude, the first approach is more efficient when P is correctly executed, while the second approach is more efficient when the use of the redundant component Q is required. This last design approach can for example be chosen if we know that the execution of the redundant component requires a supplementary duration to which any further restoring duration must be added, due to real-time constraints.

Exercise 18.8. EDC in a RAM

1. Matrices G and H:

$$G = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}, H = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

Coding operation:

$[y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, y_9, y_{10}, y_{11}, y_{12}] = [u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8] \cdot G$.
 For example, if $U = [0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1]$, then $Y = [1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1]$.

2. Error detection and correction. We suppose that bit w_6 is erroneous, and we perform:

$$S = H \cdot W^T, \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \cdot [110100111011]^T = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

The decimal value of this syndrome vector indicates the erroneous bit: bit 6. The correction is then a simple binary complementation.

3. Implementation of this code in the MMU.

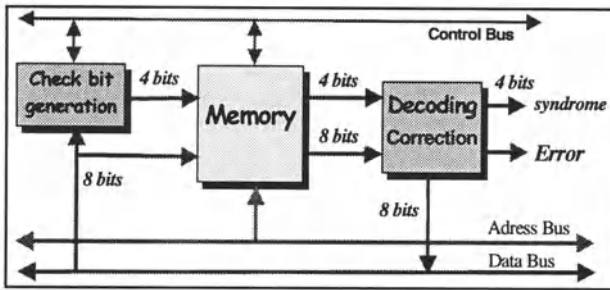


Figure E.45. Detection and correction circuit

Figure E.45 shows the structure of the EDC circuitry for this code. The ‘check bit generation’ module implements in hardware the XOR expressions to generate the 4 redundant bits. The ‘decoding and correction’ module implements the matrix product $S = H \cdot W^T$. This module uses the result S to correct the erroneous bit, and it communicates with the external system (e.g. a CPU) for *error logging*.

4. Scrubbing operation

As said in Chapter 18, the scrubbing is an off-line operation which write corrected erroneous word, and read them again, in order to check if the faults are *hard* or *soft*. If they are soft, the word has been cleaned up. On the contrary, the fault is hard and cannot be cleaned.

The previous structure is entirely compatible with such useful function.

Glossary

1. ACRONYMS

ABS	Antilock Braking System
ATE	Automatic Test Equipment
ATPG	Automatic Test Pattern Generation
BCH	Bose Chauduri Hocquenghem
BIST	Built-In Self-Test
BIT	Built-In Test
BITE	Built-In Test Equipment
BNF	Backus-Naur Form
C/DC	Condition/Decision Coverage
CAM	Computer Aided Maintenance
CAN	Control Area Network
CIRC	Cross-Interleaved Reed-Solomon Code
CMOS	Complementary MOS
COTS	Components Off The Shelf / Commercial Off-The-Shelf
CRC	Cyclic Redundancy Check
DAT	Digital Audio Tape
DFG/PFG/SFG	Deterministic / Probabilistic / Statistical Fault Grading
DFT	Design For Testability
DRC	Design Rule Checking
DUT	Device Under Test

ECC	Elliptic Curve Cryptography
EDC / ECC	Error Detecting Codes / Error Correcting Codes
EMC	Electro-Magnetic Compatibility
ESF	Extended Super Frame
FMEA	Failure Modes and Effects Analysis
FMECA	Failure Modes and Effects and Criticality Analysis
FPGA	Field Programmable Gate Array
FRC	Functional Redundancy Checking
FSM	Finite State Machine
FTM	Fault Tree Method
GSM	Global System for Mobile communication
HDB	High Density Bipolar (signal coding)
HDL	Hardware Description Language
IC	Integrated Circuit
JTAG	Joint Test Action Group
LFSR	Linear Feedback Shift Register
LRC / VRC	Longitudinal / Vertical Redundancy Check
LSB / MSB	Least/Most Significant Bit
LSSD	Level Sensitive Scan Design
MC/DC	Modified Condition/Decision Coverage
MDT	Mean Down Time
MOS	Metal Oxide Semiconductor
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure
MTTFF	Mean Time To First Failure
MTTR	Mean Time To Repair
MUT	Mean Up Time
NMR	N-Modular Redundancy
NRZ	Non-Return to Zero
PCB	Printed Circuit Board
PLA	Programmable Logic Array
PLC	Programmable Logic Controller
PLD	Programmable Logic Device
PSA	Parallel Signal Analyzer
RAID	Redundant Array of Independent Disks

RAM	Random Access Memory
ROM	Read Only Memory
RSA	Rivest Shamir Adleman
SCC	Self-Checking Checker
SOC	System On Chip
STG	State Transformation Graph
STIL	Standard Tester Interface Language
TAP	(Boundary Scan) Test Access Port
TMR	Triple Modular Redundancy
VAN	Vehicle Area Network
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VXI	VME eXtensions for Instrumentation

2. KEYWORDS

Word	Meaning	CH
acceptability curve	Curve expressing the acceptable risk rate of failures from their seriousness	17.1
acceptable product	A product whose failures have acceptable risk rates	17.1
acceptable risk rate	See <i>risk: acceptable rate</i>	
acceptance test	See <i>test: acceptance</i>	
activation: initial	The occurrence of a first error provoked by a fault. This error is called <i>primitive error</i> or <i>immediate error</i> . See also <i>fault activation</i>	4.1 13.2
active fault tolerance	See <i>fault tolerance: active</i>	
Ad Hoc approach	See <i>DFT: ad hoc approach</i>	
adaptive sequence	See <i>sequence: adaptive</i>	
adaptive vote	See <i>vote: adaptive</i>	
aggression	See <i>fault: external</i>	

alias	An alias occurs when a faulty circuit test output response gives a signature which is identical to the fault-free signature (used in BIST techniques by LFSR signature analysis)	14.5
alpha test	See <i>test: alpha</i>	
alternate	Redundant module (version) having the same specification (or a degraded form) and, often, a different implementation than the original functional module	18.4
ambiguity	An element which leads to several meanings	9.3
analysis: criticality	See <i>criticality analysis</i>	
analysis: dynamic/static	See <i>dynamic & static analysis</i>	
assertion	Functional redundancy used for software verification. It tests the validity of a property each time a given circumstance could violate it. It can be used: during the creation stages for fault removal (Ch. 10), or during the operation stage for fault detection by on-line testing (Ch. 16)	10.5 16.3
ATE	Automatic Test Equipment	12.1
ATPG	Automatic Test Pattern Generation: automatic generation of lists of test inputs and expected outputs to perform product testing	12.3
attributes of dependability	Criteria enabling the system dependability to be assessed. The most used attributes are: <i>reliability, availability, maintainability, testability, safety</i> and <i>security</i>	1.4 7
attributes of module	The behavior of a module is characterized by a set of attributes whose values define the <i>states</i> of the module	2.3
availability	It is the probability that the system is operational at the time t , knowing that it functions correctly at time 0	7.5
availability: instant	Value of the availability at a given time t : $A(t)$	7.5
availability: permanent	In permanent stage, availability value of $A(t)$ when $t \rightarrow \infty$	7.5
backward fault analysis	Step of structural fault coverage method which determines the faults detected by a given test vector by a backward process (from the outputs towards the inputs)	13.3
backward propagation or tracing	See <i>propagation: backward</i>	
backward recovery	Fault-tolerance technique which consists in bringing the system back in a state previously reached before the system execution resumption. This technique makes often use of context saving and restoring mechanisms (such as the <i>recovery cache</i>). The execution of M is resumed at a <i>recovery point</i>	18.3

bathtub curve	Reliability model which represents the evolution with time of failure rate of electronic components. Typically, it shows 3 parts: <i>infant mortality</i> where the failure rate decreases, <i>useful life</i> where the failure rate is constant, and <i>wearout</i> where the failure rate increases.	7.2
behavior	Reaction of a system mainly described as changes of states in this book	2.3
behavioral level/model	A design step/model of the system specifying its behavior	2.2 2.3
benign	See <i>failure: benign</i>	
beta test	See <i>test: beta</i>	
BIST	Built-In Self-Test. Group of <i>Design For Testability</i> methods which incorporate the test functions into the circuit	14.5
BIST: signature	Group of BIST techniques using a <i>test sequence generator</i> (usually a LFSR), a <i>compaction function</i> (usually a PSA), and a <i>signature analysis function</i>	14.5
BIT	Built-In Test. Group of <i>Design For Testability</i> methods which incorporate test facilities and offer a test interface	14.4
bit stuffing	Fault detection technique applied to data. After a number of bits with the same polarity, an additional bit is introduced with an opposite polarity. Used for instance in the CAN Bus	18.7
BITE	Built-In Test Equipment. All maintenance functions of a system	14.4
boundary scan	Scan technique belonging to the BIT design for testability. Normalized as the IEEE Standard 1149.1	14.4
branch test	See <i>test: branch</i>	
bridging fault	A particular case of <i>short</i> electronic fault See also <i>fault: short-circuit</i>	5.2
BSDL	Boundary Scan Description Language	14.4
bug	See <i>fault: structural</i> (for software technology)	3.2
burn-in test	See <i>test: burn-in</i>	
C/DC	See <i>test: Condition/Decision</i>	
CAM	Computer Aided Maintenance	12.1
CAN Bus	Control Area Network bus. Initially created for automotive industry. Normalized under ISO 11898	18.7
catastrophic	See <i>failure: catastrophic</i>	
checker	Module used in self-testing systems to detect the occurrence of errors from the observation for instance of some EDC code variables See also <i>self-checking checker</i>	16.3
checkerboard	See <i>test: memory</i>	

checksum	See <i>code: checksum</i>	
clarity	The text or the model describing a system is easy to read	9.3
client	Entity or person expressing requirements or specifications to ask for an expected product	2.2
code	Set of the codewords in an EDCC	15.1
code preserving	See <i>Totally Self-Checking System</i>	
code: m-out-of-n	Unidirectional code such that each codeword has exactly m bit '1' and $(n-m)$ bits '0'	15.4
code: two-rail	Special case of m-out-of-n code such that the codeword is obtained by adding to the word to be coded its complemented copy Also called <i>double-rail</i>	15.4
code: arithmetic	Category of codes dealing with detection and correction of errors in arithmetic systems	15.5
code: Berger	One of the codes dedicated to unidirectional errors. The redundant part expresses in binary the number of bits '0' in the word to be coded	15.4
code: bidimensional	Code applied to blocks of words Also called <i>product code</i> \Leftrightarrow <i>unidimensional</i>	15.3
code: capacity	Number of codewords that can be made with a given code Also called <i>power of expression</i> or <i>cardinality</i>	15.2
code: cardinality	See <i>code: capacity</i>	
code: checksum	Bidimensional arithmetic code based on the sum without remainder of the words of a block	15.5
code: CIRC	Cross-Interleaved Reed-Solomon Code	15.3
code: cost	Number of bits (n) of the codewords	15.2
code: coverage rate	Ratio of the number of errors detected and/or corrected by the code and the number of errors belonging to the considered error model	15.2
code: CRC	Cyclic Redundancy Check code are cyclic EDCC codes	15.3
code: cyclic	Family of linear EDCC codes. Modeled with polynomials	15.3
code: density	Ratio of the capacity of a redundant n -bit code and the theoretical number of words that can be made ($2n$)	15.2
code: disjoint	See <i>self-checking checker</i>	
code: ECC	Elliptic Curve Cryptography codes based on elliptic curves	15.1
code: EDCC	<i>Error Detecting and Correcting Code</i> . Redundant coding of information used to detect and/or correct errors	15.1
code: error corrector	See <i>code: EDCC</i>	

code: error detector	See <i>code: EDCC</i>	
code: fault secure	See <i>totally self-checking system</i>	
code: Fire	Cyclic code addressing burst multiple errors	15.3
code: linear	EDCC using multiple parity. Modeled with matrices	15.3
code: low-level: HDB	<i>High Density Binary</i> . Signal level coding	15.1
code: low-level: Manchester	Signal level coding	15.1
code: low-level: NRZ	<i>Non-Return to Zero</i> . Signal level coding	15.1
code: modulo 9 proof	An example of arithmetic code	15.5
code: multiple parity	Code using several redundant bits which are obtained by XOR combinations of some bits of the word to be coded	15.3
code: power of expression	See <i>code: capacity</i>	
code: product	See <i>code: bidimensional</i>	
code: redundancy rate	Ratio of the number of added bits ('redundant bits') and the number of bits of the word to be coded (called 'useful bits')	15.2
code: residual	Category of codes intended to detect errors in arithmetic circuits	15.5
code: RSA	Rivest Shamir Adleman codes based on the factorization of large numbers	15.1
code: self-checking	See <i>self-checking checker</i>	
code: separable / non-separable	Properties of redundant codes <i>separable code</i> : the information to be coded is explicitly included in the codeword (the codeword is made by adding redundant bits to the information data)	15.2
code: single parity	Code using one redundant bit which is the XOR of all other bits of the word to be coded	15.3
code: syndrome	See <i>syndrome</i>	
code: totally self-checking	See <i>totally self-checking system</i>	
code: two-rail	Particular case of <i>m-out-of-n</i> code Also called <i>double-rail</i>	15.3

code: unidimensional	Code applied to individual words ◊ <i>bidimensional</i>	15.3
code: unidirectional	Category of codes intended to detect unidirectional errors, i.e. multiple errors that modifies the altered bits in the same way (all '0' to '1', or all '1' to '0')	15.4
codeword	Coded element of information	15.2
cold standby redundancy	See <i>redundancy: cold standby</i>	
compaction	See <i>test: compaction</i>	
compaction function	See <i>BIST: signature</i>	
compatibility of a product	The service delivered by the product is greater than the one expected from the specifications ◊ <i>incompatibility</i>	4.2
compensation technique	Fault tolerance technique using passive redundancy such as the TMR. Does not require error detection See also <i>error masking</i>	6.4 18.2
complete diagnosis sequence	See <i>complete distinguishing sequence</i>	
complete distinguishing sequence	Diagnosis sequence which split the fault model into classes of <i>system equivalent faults</i>	12.2 13.4
completeness	All possible cases are handled	9.3
compliance test	See <i>test: compliance</i>	
component	Structural entity of a system Also called <i>module</i> or <i>sub-system</i>	2.3
composition relationships	Relationships composing sub-systems to express the structural model of a system	2.2
compositional hierarchy	See <i>hierarchy: composition</i>	
comprehension	The understanding of the semantics of a text or a model describing a system or other pieces of information	9.3
computer aided maintenance	CAM. Tools which assist the maintenance team	12.1
concision	The text or the model describing a system does not contain useless verbiage	9.3
condition	Boolean expression which does not contain any Boolean operator (AND, OR, NOT) See also <i>decision</i>	13.6

condition/ decision test	See <i>test: condition/decision</i>	
conditional maintenance	See <i>maintenance: preventive</i>	
confidentiality	Non occurrence of unauthorized disclosure of information	7.7
confinement	See <i>error confinement</i>	
consequences of faults/failures	External effects of faults or failures on the product's mission. These effects are generally classed into groups according to their seriousness: <i>minor</i> or <i>benign</i> (see <i>failure: minor</i>), <i>significant</i> (see <i>failure: significant</i>), <i>serious</i> (see <i>failure: serious</i>), <i>catastrophic</i> or <i>disastrous</i> (see <i>failure: catastrophic</i>),	4.2 17.1
consistency	No conflicts exist between definitions of the elements of a system or of a text	9.3
consistency operation	(or <i>justification</i>) One of the four basic steps of path sensitizing test generation method which verifies that the local constraints can be satisfied in the whole circuit	13.2
contamination	See <i>error propagation</i>	
continuity of service	See <i>reliability</i>	
continuity test	See <i>test: continuity</i>	
continuous on- line testing	See <i>on-line testing</i>	
contract	Document produced during the specification phase, which formalizes the mission of the product (function and duration), non-functional constraints on the environment, and the dependability attributes	2.2
control flow	Finite State model derived from a program, expressing the sequencing of the statement block and taking the input events and the internal decisions into account	13.6
control path	Path of a control flow	13.6
controllability	Ease of reaching a given state of a system behavior by exercising its inputs	6.3 14.1
corrective action	Action taken to eliminate the causes of an existing nonconformity, defect (fault) or other undesirable situation in order to prevent recurrence	2.2
corrective maintenance	See <i>maintenance: corrective</i>	
coverage	See <i>fault coverage</i> and <i>code: coverage rate</i>	
coverage table	See <i>fault table</i>	
CRC	See <i>code: CRC</i>	

creation process	See <i>development process</i>	
criticality analysis	Methods used to estimate the risks of the failures of a product	17.1
criticality level	Measurement or classification based on acceptable rate risk Also see <i>consequences of faults/failures</i>	17.1
curative maintenance	See <i>maintenance: corrective</i>	
dangerous	See <i>failure: serious</i>	
debugging	The process of detecting, locating, and correcting faults and errors. Belongs to fault removal	12.1
decision	Combination of <i>conditions</i> or <i>decisions</i> using Boolean operators (AND, OR, NOT)	13.6
decreasing reliability	See <i>reliability: decreasing</i>	
defect	See <i>fault: structural</i> (for hardware technology)	3.2
degradation	Degradation of the service delivered by a product affected by faults	18.5
delivered service	See <i>service delivered</i>	
dependability	The dependability of a system is that property of the system such that reliance can justifiably be placed on the service it delivers	1.2
dependability assessment	Techniques to measure or estimate the dependability, thanks to attributes: reliability, availability, maintainability, testability, safety, security These attributes are evaluated at three levels in the life cycle: <i>specification</i> and <i>forecast</i> assessment during the creation stages, and <i>exploitation</i> assessment during the exploitation stage There are two groups of techniques: <i>quantitative approach</i> and <i>qualitative approach</i>	7.1
dependability assurance	Set of actions justifying the reliance placed in a given product	6.5
design	Step of the life cycle which transforms specifications into a system	2.2
design for testability	See <i>DFT</i>	
design guide	Fault prevention techniques, relative to the design process, advising the design process choices	10.3
design level	Design is traditionally classified into three modeling steps: behavioral, structural, and technological	2.2
design rule checking	DRC. Example: ensure all geometric features laid out on each mask meet size, spacing, and overlap rules	11 14.4
design test	See <i>test: design</i>	

design: extraction	Fault removal technique: verification with the specifications, by reverse transformation, e.g. identification of the electronic structure from the layout	10.4
design: proof	Fault removal: formal verification technique with the specifications, by reverse transformation, or demonstration of a property of the system behavior	10.6
designer	The entity or person which creates a system or a product from requirements or specifications	2.2
destructive test	See <i>test: destructive</i>	
detection test	See <i>test: detection</i>	
Deterministic Fault Grading	See <i>fault grading: deterministic</i>	
development process	The process that leads from the specification to the product. In this book, it groups together specification, design and production Also called <i>creation process</i>	2.2
device under test	See <i>DUT</i>	
DFT	<i>Design For Testability</i> . Set of design techniques increasing the controllability and observability of the product. Used for off-line testing. There are four DFT main approaches: Ad Hoc techniques, specific design for testability, <i>Built-In Test (BIT)</i> , and <i>Built-In Self Test (BIST)</i>	14.1
DFT: ad hoc approach	Guidelines used during or after the design to facilitate the test	14.2
DFT: specific design	Group of <i>Design For Testability</i> methods which provide products naturally easy to test	14.3
diagnosis	Process of identifying the fault, if one exists See also <i>test: diagnosis, fault: localization</i>	6.3 10.5 12 13.4
diagnosis algorithm	Definition of modeling tools used to express the pieces of information handled during the diagnosis (such as fault tree), and the tasks and steps to be done to diagnose the faults Also called <i>diagnosis process</i>	12.1 13.7
diagnosis fault tree technique	Technique which successively split the set of the faults of a fault model into fault classes in order to diagnose the causes of a failure	12.2
diagnosis process	See <i>diagnosis algorithm</i>	
diagnosis test	See <i>test: diagnosis</i>	
diagnosis testing: adaptive	Diagnosis testing using an adaptive sequence	12.2

diagnosis testing: fixed	Diagnosis testing using a fixed sequence	12.2
diagnosis tree	Graphic tool allowing to determine which fault (or faults) is (are) present, from the output values produced by a system submitted to a test sequence	12.2 13.3
diagnosis: based on deep knowledge	See <i>diagnosis: model based approach</i>	
diagnosis: based on structure and function	See <i>diagnosis: model based approach</i>	
diagnosis: empirical associations	See <i>diagnosis: experimental approach</i>	
diagnosis: experimental approach	Diagnosis methods based on knowledge of relationships between possible faults or errors and the related failures Also called <i>empirical associations</i> , or <i>surface</i> or <i>shallow reasoning</i> , or <i>reasoning by associations</i>	12.1
diagnosis: model-based approach	Diagnosis methods which do not use fault or error models. The failures are diagnosed thanks to the system model Also called <i>diagnosis based on deep knowledge</i> , or <i>diagnosis based on structure and function</i>	12-1
diagnosis: reasoning by associations	See <i>diagnosis: experimental approach</i>	
diagnosis: shallow reasoning	See <i>diagnosis: experimental approach</i>	
diagnosis: surface reasoning	See <i>diagnosis: experimental approach</i>	
disastrous	See <i>failure: catastrophic</i>	
discontinuous on-line testing	See <i>on-line testing</i>	
disruption	Modification of the correct state which cause an error Also called <i>error</i> in error detecting and correcting code theory See also <i>failure: disruptive</i>	15.1
disruption operator	Operator combining a correct state and a disruption to express an error	15.1
distance: arithmetic	Fundamental notion similar to the <i>Hamming distance</i> , used to study arithmetic codes	15.5

distance: Hamming	See <i>Hamming distance</i>	
distinguishing sequence	Test sequence able to decide which fault of a given fault model is present in a circuit. Used for <i>diagnosis</i>	12.2 13.4
disturbance	See <i>fault: external</i>	
domain: dangerous	See <i>safety: dangerous domain</i>	
domain: functional	See <i>functional domain</i>	
domain: safe	See <i>safety: dangerous domain</i>	
domino effect	Cascade phenomenon occurring during the restoration of the context of a multiple task system, when using a <i>recovery point</i> fault tolerant technique	18.3
double-duplex	Fault tolerance technique using active redundancy	18.7
dreaded event	Impairments of dependability (faults, errors, failures) studied by qualitative dependability assessment techniques. Often limited to dangerous events	7.1
duplex	Self-testing technique based on structural redundancy. The main module is functionally duplicated, the outputs of the two duplicates being compared by a checker	16.3
duplicate	Duplicates redundant modules are Versions having the same implementation (used in fault-tolerance techniques) Also called <i>replica</i>	18.2
duration of the mission	Objectives of the product in terms of operational life	2.1
DUT	Device Under Test. Product connected to a tester	12.1
dynamic analysis	Group of techniques relevant to fault removal carried out by executing products or models Also called <i>test</i>	6.3
ECC	See <i>code: ECC</i>	
EDC / ECC	Error Detecting Codes / Error Correcting Codes See <i>code: EDCC</i>	15.1
embedded core	See <i>IEEE P1500</i>	
emergence	Operator determining the behavioral part of a component which effectively intervenes in the global behavior of a system	8.2
emergent functionality	The functional part of the product which is really used in the context of the mission	4.2
empirical associations	See <i>diagnosis: experimental approach</i>	

environment	<i>functional</i> : see <i>user</i> <i>non-functional</i> : entities external to the couple Product-User and having an influence on the delivered service	2.1
equivalence: mutated program	A mutated program is equivalent to the initial program if the mutation does not modifies the behavior	13.8
equivalent fault	See <i>pattern equivalent fault</i> and <i>system equivalent fault</i>	
error	An error occurs in a module (or component) when its actual state deviates from its desired or intended state	4.1
error confinement	Methods and techniques used to limit the error propagation to a certain subset of the system See also <i>fault: contention</i>	6.4 18.6
error contamination	See <i>error propagation</i>	
error detecting and correcting codes	Redundant coding to detect and/or correct errors	6.4 15.1
error detection and correction	Group of techniques of fault tolerance	6.4
error diffusion	See <i>error propagation</i>	
error logging	See <i>log file</i>	
error masking	Group of techniques of fault tolerance which does not require error detection, as the faults effects are masked See <i>compensation techniques</i>	6.4 18.2
error model	See <i>model: error</i>	
error propagation	Mechanism which transforms an error occurring in a product into one or several other errors or failures The <i>propagation</i> is conducted through one or several <i>error propagation paths</i> Also called <i>error diffusion</i> or <i>contamination</i>	4.1 15.6
error typology	See <i>model: error</i>	
error: asymmetric	<i>Asymmetric error</i> has a different probability to produce a '1' value and a '0' value	5.2
error: burst	<i>l</i> -order multiple error model such as all the errors affect a sequence of <i>l</i> consecutive bits	15.1
error: dynamic	A <i>dynamic error</i> provokes transient undesirable states (e.g. a transient oscillation on a line) Also called <i>transient error</i>	4.1
error: generic	Error associated with a modeling tool ↔ <i>error: specific</i>	6.3
error: hard	See <i>error: permanent</i> (for electronic components)	5.2

error: immediate	See <i>activation: initial</i>	
error: logical	<i>Logical error</i> is characterized by transformations of logical values: '0' becomes '1' and vice versa. <i>Non-logical error</i> provokes alterations of the logic levels outside the specification domains	5.2 15.1
error: multiple	<i>Multiple error</i> disturbs the functioning of several elements (e.g. a problem in the electrical supplying affects all the components)	5.2
error: order	Order of multiplicity. The number of elements altered by a multiple error	5.2 15.1
error: packet	Multiple error where all modified bits are grouped within a certain distance	15.1
error: permanent	A <i>permanent error</i> affect a module for a long duration (e.g. the output of a module is stuck-at '0') <> <i>error: temporary</i>	4.1
error: primitive	See <i>activation: initial</i>	
error: single	<i>Single error</i> affects one element (for instance a transistor) of the structure of the system	5.2 15.1
error: soft	Temporary error induced by transient faults in electronic components	5.2
error: specific	Error associated with a particular system <> <i>error: generic</i>	6.3
error: static	A static error provokes a stable undesirable state (e.g. a false signal '1' instead of the right one '0')	4.1
error: symmetric	<i>Symmetric error</i> provokes with the same probability, a state changing (for instance, '0' to '1') and conversely	5.2
error: temporary	A <i>temporary error</i> has limited operation duration <> <i>error: permanent</i>	4.1 5.2
error: transient	See <i>error: dynamic</i>	
error: unidirectional	<i>Gate level</i> : multiple logical error such as all altered lines are stuck at the same value <i>Code theory</i> : multiple error which modifies several bits of a word in the same sense: 0 to 1, or 1 to 0	5.2 15.4
event tree	Tree connecting correct (states) or incorrect (faults, error, failures) events with logical operators (AND, OR). Used for deductive approach in qualitative dependability assessment See <i>Fault Tree Method</i>	7.11
evolutionary maintenance	See <i>maintenance: preventive</i>	

exception mechanism	Software on-line error detection/handling/propagation mechanism	14.2 17.2
execution path	A control path which can be run when the program is executed	13.6
exploitation	See <i>operation</i>	
exploitation value	See <i>dependability assessment</i>	
exponential law	The simplest reliability model used for electronic components	7.2
extraction	Electronic extraction of an IC. Analysis which gives a transistor-level description from a mask-level layout description	10.4
extremely improbable	See <i>risk</i>	
extremely rare	See <i>risk</i>	
fabrication	See <i>production</i>	
fail-fast system	A <i>fail-fast system</i> is a <i>fail-safe system</i> which integrates a maximal duration to reach the safe state in their specifications	17.2
fail-safe system	System integrating techniques to reduce or avoid the occurrence of failures considered as catastrophic or dangerous	6.4 17.1
fail-silent system	Fail-safe technique: when an error or a failure occurs, the system turns itself into a safe 'off' or 'passive' mode which does not act on the environment Also called <i>fail passive</i>	17.2
failure	A failure occurs when the delivered service no longer complies with the specifications Taking the <i>mission</i> notion into account, a failure is the non-performance or inability of the system or component to perform its intended function for a specified time under specified environmental conditions	3.1
failure mode	Abstract viewpoint about failures, independently of the particular system functions and failures. Often defined by three parameters (value/timing, persistent/temporary, consistent/inconsistent) completed by the seriousness and risk	3.1
failure rate (λ)	Mathematical estimator of reliability which expresses a failure occurrence probability per hour, e.g. 10.6 fault/H (non MKSA unit)	7.2
failure: benign	<i>Failure</i> which has no serious consequences on the mission which carries on normally. Failure leading to upset of the users, and/or a partial reduction of the functionality of the product Also called <i>minor</i>	4.2 17.1
failure: Byzantine	See <i>failure: inconsistent</i>	3.1

failure: catastrophic	Failure leading to human loss, destruction of the product or the environment, including the controlled process Also called <i>disastrous</i>	4.2 17.1
failure: consistent	Failure perceived similarly by all users	3.1
failure: crash	Persistent <i>omission failure</i>	3.1
failure: dangerous	See <i>failure: serious</i>	
failure: disastrous	See <i>failure: catastrophic</i>	
failure: disruptive	A failure caused by a technological fault Also called <i>disruption</i>	3.2
failure: dynamic	The temporal characteristics of the product behavior are not in accordance with the specifications: e.g. response time incorrect, too fast or too slow Also called <i>timing failure</i>	3.1
failure: extremely improbable	See <i>risk</i>	
failure: extremely rare	See <i>risk</i>	
failure: impossible	See <i>risk: impossible</i>	
failure: inconsistent	The users do not perceive in the same way the failure occurrence. Also called <i>Byzantine failure</i>	3.1
failure: major	See <i>failure: significant</i>	
failure: minor	See <i>failure: benign</i>	
failure: omission	A specific <i>stopping failure</i> when no values are delivered	3.1
failure: persistent	The provided service is not in accordance with the specification, during a long period in regards with the mission duration	3.1
failure: probable	See <i>risk</i>	
failure: rare	See <i>risk</i>	
failure: serious	Failure whose negative effects on the user or the environment are quite important, the security margins being dangerously reduced. Leads to a small number of casualties and/or serious injuries of the users, and/or a serious reduction of the functionality of the product Also called <i>dangerous</i>	4.2 17.1

failure: seriousness class	Set of failures having the same seriousness	17.1
failure: severity or seriousness	Measurement of the consequences of a failure on the system, user, and environment	4.2 17.1
failure: significant	Seriousness of a failure: the mission is disturbed and the efficiency of the delivered service is reduced. Leads to injuries of the users, and/or a partial reduction of the functionality of the product Also called <i>major</i>	4.2 17.1
failure: static	A product has a static failure when, at a given time, its actual perception via its inputs, or its actual reaction are not in accordance with its specifications Also called <i>value failure</i>	3.1
failure: stopping	When the product's activity no longer evolves, a constant value being delivered to the user.	3.1
failure: systemic	A failure caused by a functional fault.	3.2
failure: temporary	The provided service is not in accordance with the specifications at a given time and for a short duration	3.1
failure: timing	See <i>failure: dynamic</i>	
failure: value	See <i>failure: static</i>	
false alarm	Generally associated with built-in test. It is an indication of a failure in a system where no failure exists.	16.3
fault	Adjudged or hypothesized cause of a failure. Also see <i>fault: structural</i> Also called <i>defect</i> (hardware) or <i>bug</i> (software)	3.2
fault activation	Raising of an error from a fault In particular, it is the first step of path sensitizing test generation methods which transforms a fault into a primitive error to be propagated to the primary outputs of the system under test	4.1 13.2
fault avoidance	Fault prevention + fault removal	6.1
fault collapsing	Technique to reduce the run time of fault simulation by identifying equivalent faults and simulating only one fault for each class	12.3
fault contention	Technique to prevent errors due to faults in a module to reach other modules of the system See also <i>error confinement</i>	6.4
fault correction	Operation which suppress present faults	6.3
fault coverage	Percentage of potential faults of a given fault model that are detected during a test	12.2 13
fault detection	Operation which highlight the presence of faults	6.3

fault diagnosis	Operation which identifies the faults altering a product	6.3
	Also called <i>fault localization</i> or <i>fault isolation</i>	12.2
		13.4
fault dictionary	List of faults, their activation, and their effects (as errors or failures), which can aid in the determination of probable causes during failure analysis of defective devices	12.3
fault forecasting	Estimation of the presence of faults (number and seriousness) Developed in Chapter 7	1.3
fault grading	Measure of how effective a set of test vectors is at detecting potential faults. Finding of the coverage of a given test sequence Also called <i>test validation</i>	12.3
fault grading: deterministic	The DFG is a simulation method which compares the results of a faulty design (fault injected) with the outputs coming from the design. It includes various simulation algorithms, such as grouping of equivalent faults, also known as <i>fault collapsing</i> , and making use of customized hardware platforms (<i>accelerators</i>)	12.3
fault grading: fault simulation	There are three approaches to fault grading, based on <i>fault simulation</i> techniques: <i>probabilistic</i> (PFG), <i>deterministic</i> (DFG), and <i>statistical</i> (SFG)	12.3
fault grading: probabilistic	The PFG is a simulation method which provides an estimation of the fault coverage rather than an exact determination. The principle is based on an analysis of the node activity in terms of <i>controllability</i> and <i>observability</i>	12.3
fault grading: statistical	The SFG reduces the cost of DFG by applying deterministic fault simulation to a sub-sets of the potential faults of the given fault model. It provides a close approximation of the DFG results, while requiring only a small fraction of the run time	12.3
fault grading: structural approach	Method evaluating the fault coverage of a test sequence by structural analysis of the product. It consists in: <i>forward simulation</i> , and <i>backward fault analysis</i>	13.3
fault injection	Technique consisting in adding faults to a system in order to analyze its behavior. Used for fault grading or to assess fault tolerance mechanisms	7.9
		12.3
fault logging	Recording of errors occurring in a product during operation in order to facilitate ulterior maintenance See also <i>instrumentation</i> and <i>logfile</i>	16.3
fault masking	Fault belonging to a passive redundant element that cannot be detected from the outside of the product. Use of compensation mechanisms	13.3
fault model	See <i>model: fault</i>	
fault prevention	Aims at reducing the creation or occurrence of faults during the system life cycle Developed in Chapters 9, 10, and 11	1.3
		6.2

fault removal	Aims at detecting and eliminating existing faults, or to show the absence of faults Developed in Chapters 12, 13, and 14	1.3 6.3
fault secure	See <i>totally self-checking system</i>	
fault simulation	Technique used for dependability assessment. Fault grading technique which provides a list of faults detected by a given test sequence (hence, the <i>fault coverage</i>) by means of a simulation program with fault injection There are four main approaches: <i>serial, parallel, deductive, and concurrent</i>	7.9 12.3
fault table	Table showing the faults covered by each vector of a test sequence Also called <i>coverage table</i>	12.2
fault tolerance	Aims at guaranteeing the service provided by the product despite the presence or appearance of faults Developed in Chapters 16, 17, and 18	1.3 6.4
fault tolerance: active	Approach that makes use of error detection and handling	18.5
fault tolerance: compensation	See <i>compensation technique</i>	
fault tolerance: passive	Approach that does not make use of error detection	18.5
fault tree method	See <i>FTM</i>	
fault tree: diagnosis	See <i>diagnosis fault tree</i>	
fault: accidental	Fault which is not intentionally created ◇ <i>intentional</i>	3.2
fault: active	A fault becomes active when it provokes an error during the operation of the product ◇ <i>passive</i>	4.1
fault: bridge	See <i>bridging fault</i>	
fault: common mode	Fault caused by the same circumstances, and thus provoking the same errors/failures, of several redundant modules in a fault-tolerant system	18.2
fault: component	Fault associated with a component. Also called <i>module fault</i>	3.3
fault: conceptual	See <i>fault: functional</i>	
fault: creation	Fault occurring during specification, design and/or production phases (excluding the operation phase)	3.2

fault: delay	For electronic models. A delay fault, occurs when a signal propagating through a circuit is slower than it really should be	5.2
fault: dormant	See <i>fault: passive</i>	
fault: dynamic	See <i>fault: temporary</i>	
fault: external	Failure cause attributed to the user or the environment. Also called <i>perturbation</i> or <i>aggression</i> or <i>disturbance</i>	3.2
fault: functional	Fault due to human activities during the product life phases. The origin is the designer during the creation steps and the user during the operational step. Also called <i>conceptual</i> fault or <i>human-made</i> fault	3.2
fault: hard	Permanent fault occurring in memory circuits <> <i>fault: soft</i>	11.3
fault: hardware	See <i>fault: physical</i>	
fault: human-made	See <i>fault: functional</i>	
fault: initial activation	See <i>activation: initial</i>	
fault: intentional	Fault created deliberately <> <i>fault: accidental</i>	3.2
fault: interaction	Fault coming from the interactions of several components	3.3
fault: intermittent	<i>Temporary fault</i> due to internal causes	3.2
fault: internal	Failure cause occurring in the product or system	3.2
fault: isolation	See <i>fault: localization</i>	
fault: localization	Identification of the faults of an erroneous system Also called <i>fault isolation</i> or <i>fault diagnosis</i>	6.3
fault: masked	A fault <i>f1</i> is masked by a fault <i>f2</i> according to a given input sequence, if the occurrence of <i>f1</i> does not provoke a failure, due to the presence of <i>f2</i>	13.5
fault: module	See <i>fault: component</i>	
fault: MOS on	Fault model at MOS level: a MOS is always conducting See also <i>fault: short-circuit</i>	5.2
fault: MOS open/off	Fault model at MOS level: a MOS is always blocked	5.2
fault: observation	Ability to detect the presence of a fault through a failure occurrence	4.1
fault: operational	Fault occurring during the operational stage of the life cycle	3.3

fault: passive	The fault does not raise error; hence, it does not disturb the product's functioning. Also called <i>dormant</i> ◇ <i>active</i>	4.1
fault: permanent	Fault that persists once it has occurred (e.g. design fault) Also called <i>static fault</i>	3.2
fault: physical	Technological fault concerning hardware technology Also called <i>hardware fault</i>	3.2
fault: short-circuit	Fault model at electronic level. Particular case: <i>bridging fault</i> which provokes wired logic (OR or AND)	5.2
fault: soft	Non-permanent faults in RAM: random, non-recurring single-bit fault	11.3 18.7
fault: static	See <i>fault: permanent</i>	
fault: structural	When the internal functional faults are concerned, a fault consists in a non-adequate structure alteration Also called <i>defect</i> (hardware) or <i>bug</i> (software)	4.1
fault: stuck-at 0/1	See <i>stuck-at fault</i>	
fault: technological	Fault of the technological means (hardware / software) used to implement the product Also called <i>hardware</i> or <i>physical fault</i> for hardware technology	3.2
fault: temporal	Electronic fault due to incorrect response time of components	3.2
fault: temporary	Fault the presence of which is time bounded. The duration range is generally assumed as short Also called <i>dynamic fault</i>	3.2 5.2
fault: transient	Temporary fault due to external causes	3.2
fault: undetectable	No input test sequence can reveal the fault at the output of the system. This corresponds to passive redundancy	8.3 13.2
fault-secure	Fundamental property of a self-testing system which guarantees that no failure can occurs which is not immediately detected	16.3
feature	Element of a modeling tool or language	2.2 6.3
feature restrictions	Prevention techniques for software which consist in avoiding features which increase the fault risk (shared variables, goto, etc.)	11.2
final test	See <i>test: final</i>	
Fire code	See <i>code: Fire</i>	
FIT PLA	BIT technique used to improve the testability of PLA	14.4
fixed sequence	See <i>sequence: fixed</i>	

FMEA	<i>Failure Modes and Effects Analysis</i> : normalized technique dedicated to <i>qualitative analysis</i> of <i>reliability</i> and <i>safety</i>	7.10
FMECA	<i>Failure Modes and Effects and Criticality Analysis</i> is a variant of FMEA that associates a probability with the failure of the components and with their effects	17.1
forecasting value	See <i>dependability assessment</i>	
formal identification	See <i>test: identification</i>	
formal proof	See <i>design: proof</i>	
formal proof: deductive approach	Formal proof approach which demonstrates properties, starting from the conclusions	10.6
formal proof: inductive approach	Formal proof approach which demonstrates properties, starting from the hypotheses	10.6
formal proof: symbolic execution	A technique to implement inductive approach of formal proof by handling symbols instead of values	10.6
forward propagation	See <i>propagation: forward</i>	
forward recovery	Fault tolerance techniques resuming the system execution after an error detection in a new state (not previously reached)	18.4
forward simulation	Structural (e.g. gate level) simulation executing the model in the direct way; used for instance in path sensitizing test methods <> <i>backward propagation</i>	13.3
frequent	See <i>risk: frequent</i>	
FTM	Fault Tree Method. Deductive approach for qualitative dependability assessment See also <i>event tree</i>	7.11 10.6
FTM: basic event	Leaves of a Fault tree	7.11
full scan	See <i>scan: full</i>	
function	The <i>function</i> defines what the product is intended for and justifies its existence. An element of the <i>mission</i>	2.1
functional characteristics	See <i>mission</i>	
functional domain: dynamic	Set of possible input and/or output sequences of values as defined by the product specifications	8.2

functional domain: static	<i>Static input domain</i> : set of input values which can be applied to the product as defined by the product specifications <i>Static output domain</i> : set of output values which can be given by the product as defined by the specifications	8.2
functional environment	See <i>user</i>	
functional redundancy	See <i>redundancy: functional</i>	
functional test	See <i>test: functional</i>	
fusion	Operator combining the behaviors of several modules, taking their correlations into account	8.2
galloping	See <i>test: memory</i>	
Galois Field	Mathematical structure which has fundamental applications to Cyclic Error Detecting and Correcting Codes	15.3
guidelines	Best practices to reach an objective, for example testability improvement, fault prevention, etc.	10.3 14.2
Hamming distance	Fundamental property of redundant codes which allows the detection and/or correction of errors Number of bits that differ between two binary words	15.2
hard fault	See <i>fault: hard</i>	
HDB	See <i>code: low-levelHDB</i>	
HDL	<i>Hardware Description Language</i> . Language which describes circuits in textual code. The two most widely accepted HDLs are VHDL and Verilog	2.2
hierarchy: composition	Expression of a system as the composition of <i>sub-systems</i> or <i>components</i> which are again broken down into sub-systems	2.3
hierarchy: use	Defines a system, highlighting the services used (or called) by a component and offered by others	2.3
hot standby redundancy	See <i>redundancy: hot standby</i>	
hot swap hardware	Components (CPU/Memory, I/O boards, power/cooling modules) that can be changed or serviced while the system remains on-line	18.5
IDDQ testing	Method for enhancing the quality of IC tests by measuring the power supply current of a CMOS circuit during quiescent states Detects the physical defects that creates conduction paths between the power supply and the ground lines (e.g. stuck-on faults)	12.1
IEEE P1450	Standard Tester Interface Language (STIL). Language describing test pattern and application protocols in standard neutral form	12.1
IEEE P1500	Embedded Core Test. Application of tests to embedded cores: test-description language, test-control mechanisms and peripheral access mechanisms	12.1

IEEE Std. 1149.1.1990	IEEE standard describing the Test Access Port and Boundary Scan Architecture	14.4
IEEE Std. 1155	See <i>VXI</i>	
impairment	Opposed to dependability (degradation mechanism): fault – error – failure. Developed in Chapters 4 and 5	1.4
implementation	See <i>production</i> (for software technology)	2.2
implementation constraints	Fault prevention techniques defining implementation restriction, used for software	11.3
impossible	See <i>risk: impossible</i>	
incompatibility (of a product)	The service delivered by the product is different than the one expected from the specifications	4.2
incompleteness (of a product)	The service delivered by the product is less than the one expected from the specifications	4.2
incompleteness (specification)	Definition or properties of an object having potentially multiple meanings. Fuzziness of its semantics. Absence of pieces of information	3.3
inconsistency	Contradictory definitions or properties of one object or of several objects	3.3
increasing reliability	See <i>reliability: increasing</i>	
inertia (of the environment)	Mean time between the occurrence of a failure and the beginning of its external consequences on the mission	4.2
input sequence	See <i>sequence: input</i>	
inspection	A formal review technique based on nine steps	9.4
instrumentation	Adding of mechanisms to detect errors and record data during the operation of the product. Used to make test detection and diagnosis easier	14.2 16.3
integration test	See <i>test: integration</i>	
integrity	Non occurrence of improper alterations of information	7.7
intrinsic safety	See <i>safety: intrinsic</i>	
irredundant element	An element of a system is irredundant if its removal causes the system to be functionally different	8.3
JTAG	The Joint Test Action Group. This group created the foundation for the IEEE 1149.1	14.4
language	See <i>modeling tool</i> (generally considered as defined formally)	2.3
latency	<i>Latency</i> is the mean time between the occurrence of a fault and its initial activation as an error at the level of a given module By extension: meantime between the occurrence of a fault/error in a given module and the raising of an error in another given module	4.1

level: behavioral	See <i>behavioral level</i>	
level: logical	See <i>logical level</i>	
level: physical	See <i>physical level</i>	
level: structural	See <i>structural level</i>	
level: symbolic	See <i>symbolic level</i>	
level: technological	See <i>technological level</i>	
LFSR	<i>Linear Feedback Shift Register</i> . Synchronous sequential circuit using Flip-Flops and XOR gates, which generates a pseudo-random pattern of 0s and 1s. Used for signature analysis in BIST techniques	14.5
life cycle	Succession of the stages of a product's life: specification, design, production, operation	2.2
likelihood test	Verification of a property based on functional redundancy	16.3
link: logical	Elements interconnecting modules in a system	2.3
localization test	See <i>test: localization</i>	
log file	A file storing activities maintained to facilitate auditing and recovery (in particular fault detection) Also called <i>error logging</i>	16.3 18.7
logical level	One of the steps of the development process of a product	2.2
logical links	Define the relationships between the components of a system	2.3
logical test	See <i>test: logical</i>	
LSSD	Level Sensitive Scan Design. <i>Scan design</i> technique proposed by IBM in the 60's	14.4
maintainability	Attribute of dependability with regard to the easiness in performing the maintenance actions In a quantified way, it is the measure of the interruption duration of the service if a failure appears. A useful estimator associated with this measure is the MTTR (<i>Mean Time To Repair</i>). The term <i>serviceability</i> is also used by numerous electronic or computer manufacturers	7.4
maintenance	Actions processed on the product structure during its useful life. Contains <i>preventive, corrective maintenance, and adaptive maintenance</i>	2.2 7.4
maintenance testing	See <i>test: maintenance</i>	
maintenance: corrective	Actions applied to a product after it failed in order to restore its service Also called <i>curative</i>	2.2 7.4 12.2

maintenance: evolutive	Actions applied to a product in order to improve or modify its functionality	2.2 7.4 12.2
maintenance: in situ	Facilities integrated in the product site in order to facilitate the maintenance operation in situ	14.6
maintenance: preventive	Actions applied to a product prior to failures in order to detect the presence of faults and to correct them: <ul style="list-style-type: none"> • <i>systematic</i> (or <i>scheduled</i>) <i>preventive maintenance</i> (e.g. every 1000 hours of service) • <i>conditional preventive maintenance</i> (e.g. the maintenance is decided if the temperature is excessive) 	2.2 7.4 12.2
maintenance: remote facilities	Test facilities to detect and diagnose a product from a remote specialized center	14.6
maintenance: troubleshooting and repair	Set of actions aimed at maintaining or restoring a product in a specified state	7.4
major	See <i>failure: major</i>	
Manchester	See <i>code: low-level Manchester</i>	
manufacturing	See <i>production</i> for hardware products	
marching	See <i>test: memory</i>	
Markov model	Non deterministic state graph model used for quantitative analysis of dependability	7.9
MC/DC	<i>Modified Condition/Decision Coverage</i> . See <i>test: MC/DC</i>	
MDT	<i>Mean Down Time</i> : mean time during which the product does not deliver a service	7.5
means for dependability	To provide a product having the required dependability level, that is, the ability to deliver a service and to reach confidence in this ability	1.4
method	A detailed approach to the achieving of prescribed goals	10.3
minor	See <i>failure: minor</i>	
mission	The <i>mission</i> specifies the product's objective in terms of the <i>function</i> to perform and its <i>duration</i> . Also called <i>functional characteristics</i> of a product	2.1
model	One instantiation of a modeling tool to express a specific system	2.3
model based approach	See <i>diagnosis: model based approach</i>	
model: design level	Classical modeling level used in hardware design: behavioral, structural, technological	2.2

model: error	An <i>error model</i> defines a set of faults characterized as errors by a property on desired or intended behavior Also called <i>error typology</i>	5.1 15.1
model: fault	A <i>fault model</i> defines a set of faults characterized by physical/structural properties on the desired model structure	5.1
modeling tool	Generic means (language or notation) to express the system. The expression of a specific system is called a <i>model</i>	2.3
modified condition/ decision	See <i>test: MC/DC</i>	
module	See <i>component</i>	
module: functional	A module containing the basic functional elements ◊ <i>module: redundant</i>	8.3
module: redundant	A module containing redundant elements ◊ <i>module: functional</i>	8.3
Monte Carlo simulation	Quantitative dependability evaluation method based on simulation and fault injection	7.9
MTBF	<i>Mean Time Between Failures</i> . Maintenance indicator. The time between two failures on a piece of equipment (calculated)	7.2
MTTF	<i>Mean Time To Failure</i>	7.2
MTTFF	<i>Mean Time To First Failure</i> . It is the same as MTTF	7.2
MTTR	<i>Mean Time To Repair</i> Mean time between the instant of failure occurrence and the return of the product to full functional operation	7.4
MUT	<i>Mean Up Time</i> : mean time during which the product delivers its service	7.5
mutant	A system, such as a program, modified by a mutation	13.8
mutation	Modification of the structure of a system (generally by a fault) See <i>test: mutation</i>	13.8
need	Expectations of the product's users, that is to say knowing <i>why</i> he/she has to use a product.	2.2
netlist	Basic structural model of electronic circuits, at gate or MOS level	2.2
NMR	N-Modular Redundancy. Fault tolerant technique derived from the TMR technique, using active redundancy	18.5
non-ambiguity	An element which has only one interpretation	9.3
non-destructive test	See <i>test: destructive</i>	
non-functional characteristics	Part of the product specifications dealing with constraints on the non-functional environment and with dependability requirements	2.2

non-functional environment	See <i>environment</i>	
non-regression test	See <i>test: non-regression</i>	
non-repairable product	Product whose faults cannot be removed	6.3
notation	See <i>modeling tool</i> (generally considered as having informal semantics)	2.3
NRZ	See <i>code: low-level NRZ</i>	
N-self checking	Fault tolerant technique derived from the N-versions technique	18.7
N-versions	Fault tolerance technique, based on several duplicates of a same module, whose outputs are treated by a voter to produce the final result. TMR is a 3-Versions	18.2
observability	Ease of determining, from the outputs of a product, the current state of its behavior by exercising its inputs Complementary of <i>controllability</i> for <i>testability</i>	6.3 14.1
off-chip test	Test resources are external to the device under test Apply to off-line classical testing methods	12.1 14.5
off-line testing	Group of techniques to test a product (or a module), suspending its operational life	6.3 12.1
on-chip test	Test resources are integrated to the device under test Apply to BIST techniques	12.1 14.5
on-line testing OLT	Group of techniques to test a product (or a module) in its operation context <i>Discontinuous OLT</i> : test functions are applied at predefined instants in the life time of the product <i>Continuous OLT</i> (or <i>self-testing</i>): faults are detected as soon as they produce errors/failures	6.3 12.1 16.1 16.3
open	See <i>fault: MOS open/off</i>	
operation	Step of the life cycle which integrates the product in a given environment in order to deliver a service Also called <i>exploitation, useful life, or utilization</i>	2.2
operational lifetime	See <i>duration of the mission</i>	
optimal test sequence	See <i>test: optimal sequence</i>	
output sequence	See <i>sequence: output</i>	
parametric test	See <i>test: parametric</i>	
partial scan	See <i>scan: partial</i>	

passive fault tolerance	See <i>fault tolerance: passive</i>	
path sensitizing	See <i>test: path sensitizing</i>	
path test	See <i>test: path</i>	
path: control	See <i>control path</i>	
pattern	See <i>test sequence</i>	12.2
pattern equivalent faults	Group of faults of a fault model whose effects on the outputs of the product cannot be distinguished by the input sequence application	12.2
perturbation	See <i>fault: external</i>	
phase	Defined segment of work. Also called <i>stage</i> or <i>step</i> . A set of phases constitutes a <i>process</i>	2.2
physical level/model	Technological level/model implementing the features of the symbolic model	2.2
ping-pong	See <i>test: memory</i>	
post-condition	Functional redundancy used for on-line testing of software. It analyzes the correctness of an operation at the end of the treatment	10.5 16.3
pre-condition	Functional redundancy used for on-line testing of software. It verifies that the use context of an operation is correct	10.5 16.3
prevention	See <i>fault prevention</i>	
preventive maintenance	See <i>maintenance: preventive</i>	
prime element	An element (e.g. a gate) is said to be prime if none of its inputs can be removed without causing a functional change of the system behavior	8.3
Probabilistic Fault Grading	See <i>fault grading: probabilistic</i>	
probable	See <i>risk</i>	
process	A set of <i>phases</i> . Example: development process	2.2
process control	Techniques which apply test to the manufacturing equipment. Extended to the evaluation of any product/system development process	11.2
process: creation	See <i>development process</i>	
process: development	See <i>development process</i>	
product	Physical entity destined to satisfy needs of one or several users	2.1
product code	See <i>code: bidimensional</i>	
product: acceptable	See <i>acceptable product</i>	

product: referent	See <i>referent product</i>	
product: standard	See <i>referent product</i>	
production	Stage of the life cycle which transforms a system into the final product by hardware and/or software technological means Also called <i>manufacturing</i> or <i>implementation</i>	2.2
production testing	See <i>test: production</i>	
program mutation	See <i>test: mutation</i>	
propagation path	Trace of errors in the structure of the system during an error propagation	4.1
propagation: backward	Backward simulation of the functioning of a system from predefined output or internal values or symbols, to find the input vectors which provoke them. Used in path sensitizing structural test methods Also called <i>backward tracing</i>	13.2
propagation: forward	Simulation of the functioning of a system with values or symbols, to find constraints on the propagation of a predefined error. Used in path sensitizing structural test methods	13.2
property: behavioral	Expression of an intended property on the behavior of a system, whose violation defines an error	5.1
property: generic	Property associated with a modeling tool and not with a particular modeled system	5.1
property: physical/ structural	Expression of an intended property on the structure of a system, whose violation defines a fault	5.1
prototyping	In this book, technique which derives a basic tool from a model, to detect faults in the understanding of the model	9.3
PSA	<i>Parallel Signal Analyzer</i> . Circuit based on LFSR structure used for compaction testing in BIST techniques	14.5
qualitative assessment: deductive approach	Deduction of failures from faults or errors (dreaded events)	7.1
qualitative assessment: inductive approach	Deduction of events (faults or errors) from potential failures	7.1
quality (ISO 8402)	Totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs <i>Entity</i> : item which can be individually described and considered	1.1

quality assurance	Procedures, techniques and tools applied by professionals to ensure that a product meets or exceeds prescribed standards during a product's development cycle	11.2
quality assurance test	QA tests for electronic components: life, mechanical, thermal, lead fatigue, solderability, etc.	11.2
quality control	Analysis of samples of the production in order to determine the quality of the produced components	6.2 11.2
RAID	Redundant Array of Independent Disks. Fault tolerance technique for mass storage units using structural redundancy	18.7
rare	See <i>risk</i>	
reasonably probable	See <i>risk: reasonably probable</i>	
reasoning by associations	See <i>diagnosis: experimental approach</i>	
reconfiguration	The process for a product to automatically use alternative resources, so as to not interrupt or to resume its operation	6.4 18.5
reconvergent fan-out	Structural property of a gate circuit allowing one signal to propagate through several paths before converging towards a same component	13.3
recovery	Technique used in fault tolerance approaches consisting in reaching a correct state after an error detection See also <i>backward recovery, forward recovery</i>	18.3 18.4
recovery block	One of the <i>forward recovery</i> techniques of fault-tolerance	18.4
recovery cache	<i>Backward recovery</i> requires the implementation of execution context saving and restoring mechanisms. One of the most popular technique is named <i>recovery cache</i>	18.3
recovery point	State of the system in which the system processing is resumed during a backward recovery technique Also called <i>retry point</i> and <i>rollback point</i>	18.3
recovery: backward/ forward	See <i>backward recovery</i> and <i>forward recovery</i>	
redundancy	Presence of elements of a system which are not necessary to satisfy the normal input/output relationships (in absence of fault)	8.1
redundancy rate	Functional redundancy rate = (size (Universe) - size (Domain))/ size (Universe) For a EDC code, see <i>code: redundancy rate</i>	8.2
redundancy: active	The structural redundancy of a system is active if the design is not optimal without any possibility to directly remove any element ◊ <i>redundancy: passive</i>	8.3

redundancy: cold standby	Separable redundant modules which are in a passive state (<i>off-line</i>), waiting to be activated	8.3, 18.5
redundancy: dynamic functional domain	A dynamic functional domain of a product is redundant if it is strictly included in the dynamic functional universe of this product	8.2
redundancy: functional	Certain theoretical input values are not applicable to the product by the functional environment as defined by the specifications. Extended to the outputs and inputs/outputs values	8.2 16.3
redundancy: hot standby	Separable redundant modules which are in an active state (<i>on-line</i>) in parallel with the functional module	8.3 18.5
redundancy: off-line separable	See <i>redundancy: cold standby</i>	
redundancy: on-line separable	See <i>redundancy: hot standby</i>	
redundancy: passive	The structural redundancy of a system is passive if some elements can be removed without changing the produced behavior <> <i>redundancy: active</i>	8.3
redundancy: semantic	Presence of elements of sentences in a text whose meaning can be deduced from others sentences of the text	8.1
redundancy: separable	The structural redundancy of a system is separable if the redundant elements and the non-redundant elements are located in different modules. Thus, the system possesses a <i>functional module</i> and several <i>redundant modules</i> (<i>versions, replicates, duplicas</i>)	8.3
redundancy: structural	A system has a structural redundancy if its structure possesses some elements not necessary to produce a behavior conform to the specifications, assuming that all the structure elements provide a correct functioning	8.3 16.3 18.1
redundancy: syntactic	Presence of lexicographical or syntactical elements which are not necessary to understand the sentence's meaning	8.1
redundant functional domain	A functional static/dynamic domain of a product is redundant if it is strictly included in the static/dynamic functional universe of this product	8.2
Reed-Muller structure	Gate structure based on Galois's field to design circuits having short test sequences	14.3
reference list	Recorded test sequence	12.2
referent product	Product considered as faultless, used in a test, in parallel with the tested product. Its outputs are compared with the outputs produced by the tested product Also called <i>standard product</i>	12.2

relationship: composition	See <i>composition relationship</i>	
relationship: service	See <i>service relationship</i>	
reliability	Attribute of dependability with regard to the continuity of the service The aptitude of a product to accomplish a required function in given conditions, and for a given interval of time In a quantified way, reliability is a function of time which expresses the conditional probability that the system has survived in a specified environment till the time t , given that it was operational at time 0	7.2
reliability assurance test	Techniques used during the manufacturing process to guaranty reliability level of the produced components	11.2
reliability block diagram	Model used for quantitative analysis of reliability	7.9
reliability evaluation	Tests applied to samples of the produced circuits in order to measure or estimate the reliability parameters of this population	7.2 11.2
reliability model	Mathematical function of time expressing the evolution of the reliability of a population of components	7.2
reliability tests	Experiments applied to samples of the manufactured population: <i>curtailed, censured, progressive, progressive curtailed, with progressive constraints</i>	7.2
repair	Actions of the fault removal which restore the functioning of a product. Applied to repairable products	6.3
repair rate (μ)	Mathematical estimator of maintenance which expresses a repair probability per hour	7.4
repairable product	Product to which fault removal actions can lead to the restoration of its functionality <> <i>non-repairable</i>	2.2 6.3
replica	See <i>duplicate</i>	
requirements	Expression of the needs which justify the creation and the use of a product	2.2 9.2
retry mode	Fault tolerance technique consisting in executing again an erroneous component See also <i>backward recovery</i>	18.3
retry point	See <i>recovery point</i>	
reuse	Use of a component previously developed for another product	4.2
review	Technique used to remove faults by human analysis	9.4

risk	Occurrence probability of a failure, assessed by measurements. For example, an event is said: <ul style="list-style-type: none"> • <i>probable</i> if its occurrence probability is $> 10^{-5}$ • <i>rare</i> if its occurrence probability $\in (10^{-7}, 10^{-5})$ • <i>extremely rare</i> if its occurrence probability $\in (10^{-9}, 10^{-7})$ • <i>extremely improbable</i> if its occurrence probability is $< 10^{-9}$ 	17.1
risk acceptability	Part of the safety space (seriousness of failure, occurrence probability) in which a system is said to be acceptable in terms of safety	17.1
risk: acceptable rate	Maximum probability accepted for the occurrence of a failure. Often defined for all the failures of a seriousness class Also called <i>tolerable probability</i>	17.1
risk: frequent	A subdivision of the risk class <i>probable</i> . Probability $> 10^{-3}$	17.1
risk: impossible	Event having a very small probability of occurrence ($< 10^{-9}$) Also called <i>extremely improbable</i>	17.1
risk: reasonably probable	A subdivision of the risk class <i>probable</i> . Probability $\in (10^{-5}, 10^{-3})$	17.1
RM structure	See <i>Reed-Muller structure</i>	
robustness	Property of a system which defines its capability to provide a function which is acceptable by the user according to given perturbations. Frequently defined as the characteristic of a system which guarantees that its functionality is maintained even if specified operational and utilization requirements are violated	4.2
rollback point	See <i>recovery point</i>	
RSA	See <i>code: RSA</i>	
safety	Attribute of dependability with regard to the non-occurrence of failures of given criticality level (generally <i>catastrophic</i>) Safety is measured as the probability that the product will not have failures belonging to unacceptable seriousness classes, between the initial time and a given time t	7.6 17.1
safety class	Class of safety defined in the space: <i>seriousness</i> of failure x <i>acceptable risk rate</i>	17.1
safety: dangerous domain	Notion associated with fail-safe systems. The behavioral universe is split into: <ul style="list-style-type: none"> • the <i>dangerous domain</i> grouping catastrophic and/or dangerous failures whose occurrence is unacceptable, • the <i>safe domain</i> grouping the normal functioning and the failures whose occurrence is acceptable 	17.2
safety: intrinsic	Group of techniques constraining the development process with technological solutions which are known to be safe. These solutions essentially exploit physical properties	17.2

safety: safe domain	See <i>safety: dangerous domain</i>	
safety: structural redundancy	Structural redundancy is used in order to reduce the occurrence probability of failures belonging to dangerous safety classes	17.2
scan design	BIT Technique (DFT) which led to LSSD, and the <i>boundary scan</i> IEEE 1149.1 standard	14.4
scan domain	Part of a circuit which implement a separate scan design	14.4
scan: full	Scan technique applied to the whole product	14.4
scan: partial	Technique which implements scan design in a part of the product	14.4
SCC	See <i>self-checking checker</i>	
scenario	Input/Output sequences which simulates the interactions of a system with its environment	9.3
scheduled maintenance	See <i>maintenance: preventive</i>	
schmoo plots	Measure of the influence of parameters (supply voltage, current, frequency) on test results. Used to help the IC designer to characterize the operational regions of a device	12.1
scrubbing	Technique used to correct soft errors in dynamic RAMs	18.7
security	Attribute of dependability with regard to the prevention of unauthorized access and/or handling information Covers two parameters: <i>confidentiality</i> and <i>integrity</i>	7.7
self-checking checker	A <i>checker</i> is said to be <i>self-checking</i> with respect to a defined fault model F if it is <i>code-disjoint</i> and <i>self-testing</i> <ul style="list-style-type: none"> • <i>Code-disjoint</i>: a module transforming inputs belonging to an EDC code to an output EDC code is code-disjoint if any codeword at the inputs gives an output codeword and conversely if any non-codeword inputs gives a non-codeword outputs • <i>Self-testing</i>: expresses that every fault of F is detectable on the tested output by at least one functional input vector 	16.3
self-purging	Fault tolerance technique derived from the N-Versions with adaptive voter	18.5
self-testing	Continuous on-line testing to detect faults as soon as they produce errors Also for <i>Self-Checking Checker</i> : property of a checker such that each fault is detected at its output by application of the normal input codewords See <i>self-checking checker</i>	6.4 16.3
separable	See <i>code: separable</i>	
sequence length	Number of vectors of the test sequence	12.3

sequence: adaptive	Test sequence whose input and output values are dynamically defined, taking the previous results of the test application into account <> <i>sequence: fixed</i>	12.2
sequence: fixed	Test sequence whose input and output values are defined prior to the test processing <> <i>sequence: adaptive</i>	12.2
sequence: input	List of the inputs of a test sequence	12.2
sequence: output	List of the outputs of a test sequence	12.2
serious	See <i>failure: serious</i>	
seriousness	See <i>failure: severity or seriousness</i>	
service degradation	See <i>degradation</i>	
service delivered	The <i>delivered service</i> is the product's real behavior when placed in its applicative environment	2.1
service relationships	Relationships between sub-systems expressing that one uses services provided by others	2.2
serviceability	Measure of the ease with which a system functioning is restored to a specified state after the system is repaired. Used to express the maintainability See <i>maintainability</i>	7.4
severity	See <i>failure: severity or seriousness</i>	
shallow reasoning	See <i>diagnosis: experimental approach</i>	
short-circuit	See <i>fault: short-circuit</i>	
signature	See <i>test: signature analysis</i>	
signature analysis	Technique used in compaction test technique. The signature synthesizes the output values as the result of a likelihood property LFSR signature analysis: used for BIST off-line techniques	12.2 14.5
signature analysis function	See <i>BIST: signature</i>	
significant	See <i>failure: significant</i>	
simplicity	The concepts manipulated by a text (or a model) describing a system are simple. In particular, the number of these concepts is limited and they are loosely coupled	9.3
simulation: fault	See <i>fault simulation</i>	
simulation: Monte Carlo	See <i>Monte Carlo simulation</i>	

snapshot	Image of the system execution context at a given time. It is used for example for backward recovery technique implementation	18.3
soft fault	See <i>fault: soft</i>	
software instrumentation	See <i>instrumentation</i>	
software: structural testing	Fault removal techniques based on the program <i>control flow</i> : <ul style="list-style-type: none"> • <i>statement</i> test • <i>branch</i> and <i>path</i> test • <i>condition</i> and <i>decision</i> test (see C/DC and MC/DC) 	13.2
spare module	Redundant off-line module	8.3
specification	Stage of the life cycle which defines the characteristics of the product to be created. The result of this operation is a document called specifications or contract (see <i>contract</i>)	2.2 9.3
specification assessment value	See <i>dependability assessment</i>	
stable reliability	See <i>reliability: stable</i>	
stage	See <i>phase</i>	
standard product	See <i>referent product</i>	
standby: hot / cold	See <i>redundancy: hot standby / cold standby</i>	
state	Set of the values taken by the <i>attributes</i> of a module Internal property of a module	2.3 4.1
statement test	See <i>test: statement</i>	
static analysis	Groups of techniques of the fault removal which are made without execution of the analyzed models or products	6.3
Statistical Fault Grading	See <i>fault grading: statistical</i>	
step	See <i>phase</i>	
STIL	See <i>IEEE P1450</i>	
stochastic Petri net	Non-deterministic parallel state graph model whose arcs are labeled by probabilistic values; used for dependability assessment	7.9
strobing	Term used for test: number of times a test equipment looks at the output data of a DUT during a period	12.2
structural analysis	<i>Fault grading</i> methods which study the faultless system and deduce all the faults (of a model) that can produce failures	12.3
structural level/model	A design step/model of the system expressing it as a structured system (composed of <i>sub-systems</i> or <i>components</i> or <i>modules</i>)	2.2

structural redundancy	See <i>redundancy: structural</i>	
structural testing	See <i>test: structural</i>	
structure	Defines a system as linked components	2.3
structured-functional model	Defines a system by its structure and the behavior of its components	2.3
stuck-at fault	Fault/error model at gate level: fault that keep a circuit node (input or output) at a logical level one or zero	5.2
stuck-OFF stuck-ON	Fault/error models at MOS level <i>Stuck-On</i> : the transistor is always conducting <i>Stuck-Open</i> : the transistor is blocked in the OFF state	5.2
stuffing	See <i>bit stuffing</i>	
style guide	See <i>guidelines</i>	
sub-system	See <i>component</i>	
surface reasoning	See <i>diagnosis: experimental approach</i>	
symbolic execution	The system is executed with symbols instead of values. The results are symbolic expressions	10.6
symbolic level/model	Technological level/model taking an abstract view of the execution means	2.2
syndrome	Vector resulting from a mathematical treatment (check relations) of a codeword which allows to detect and/or correct an error from of a given codeword. This vector is equal to zero is no errors occurred	15.3
system	Set of linked <i>components</i> that act together as a whole to achieve a given mission, that is a function during a certain period of time	2.3
system equivalent faults	Group of faults of a fault model whose effects on the outputs of the product cannot be distinguished, whatever input sequence is applied Also called <i>absolute equivalent faults</i>	12.2
systematic maintenance	See <i>maintenance: preventive</i>	
TAP	The Boundary Scan Test Access Port. It is formed by the TDI, TDO, TCK, TMS and the optional TRST pin	14.4
TAP controller	A sixteen state FSM that controls the Boundary Scan logic on the IC	14.4
technological level/model	A design step/model concerning the implementation of design models using hardware/software technologies. Composed of <i>symbolic level/model</i> and <i>physical level/model</i>	2.2

termination mode	One of the forward recovery techniques of fault tolerance, which consists in completing the task started by a module P by using a redundant module Q , after the detection of an error in P	18.4
test	Dynamic techniques relevant to fault removal. It is an experiment (input sequences) applied to an executable product or model by a tester which compares the given results with expected values The process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements, or to identify differences between expected and actual results (IEEE Std 729.1983) Also called <i>dynamic analysis</i>	6.3
test application	Test processing performed by the tester which applies the test sequence to the product	12.3
test equipment	See <i>tester</i>	
test evaluation	See <i>fault grading</i>	
test generation	See <i>test pattern generation</i>	
test pattern	See <i>test sequence</i>	
test pattern generation (TPG)	Technique to determine the test sequence for a given product	12.3
test sequence	List of <i>test vectors</i> used by a tester to detect and/or diagnose faults in a product. This term is often restricted to the sequence of input vectors Also called <i>test pattern</i>	10.4 12.2
test sequence generator	See <i>BIST: signature</i>	
test sequence: quality	Two main parameters are used to evaluate the quality of a test sequence: the <i>length</i> (number of test vectors) and the <i>fault coverage</i> (percentage of the faults of a fault model which are detected)	12.2
test vector	Element of a test sequence: couple (input vector, output vector)	12.2
test with/without fault model	Test method based on the detection of faults belonging to a pre-defined fault model / without precise hypotheses about the faults	12.2
test: accelerated	Test experiment with stress constraints: elevated power supply and/or temperature	7.2 11.2
test: acceptance	Another name for final test for final checking of the product See <i>test: final</i> , <i>test: compliance</i> , <i>test: conformity</i> Also used to name on-line checking used in fault tolerance mechanism to detect errors	14.2 18.3
test: algorithmic approach	A specific ATPG handling each fault of a fault model	12.3

test: alpha & beta	Test performed by selected groups of users	12.1
test: branch	Software structural testing technique which takes the control flow branches as elements to define the test sequence coverage	13.6
test: burn-in	Production test carried out with environmental constraints such as the temperature or the electric supply. It is a non-destructive accelerated test used to detect and eliminate any defects which might appear in a product during its early life	12.1
test: C/DC	See <i>test: condition/decision</i>	
test: censured	Test used to evaluate the reliability of a population of components which stops when a given number of faults is reached	7.2
test: compaction	Test technique which reduces the data coming from the DUT by a mathematical treatment. Used in <i>signature testing</i>	12.2 14.5
test: compliance	Test to ensure the adequacy of the product with its specifications	12.1
test: condition/decision	Structural testing methods used for software program, which require that 1) each decision must take the values True and False at least once, 2) each condition must take the value True and False at least once, 3) each input and output point of the components (subprograms, etc.) must be executed at least once. The coverage rate is noted C/DC (<i>Condition/Decision Coverage</i>)	13.6
test: conformity	Acceptance test performed by the client or an external organization	12.1
test: continuity	Techniques which verifies that the connections between components are without defects: printed circuit boards, cables, connectors, etc.	12.1
test: curtailed	Test used to evaluate the reliability of a population of components whose duration is fixed a priori	7.2
test: design	Fault removal techniques based on functional test, used during design stage	6.3 10.5
test: destructive / non-destructive	A test is destructive if the tested product can be destroyed during the test process. Destructive test are employed for quality control and reliability evaluation	11.2 12.1
test: detection	Test techniques answering the question: <i>does the product function correctly?</i>	12.1 12.2
test: diagnosis	Test techniques answering the question: <i>which faults affect the product?</i> There are two main categories of diagnosis techniques: <i>fixed diagnosis</i> which uses a fixed test sequence, or <i>adaptive diagnosis</i> for which the next test vector depends on the responses given by the product to the preceding test vectors	12.1 12.2
test: exhaustive	Test technique using all input vectors to test a combinational circuit	12.3

test: final	Test applied to a complete system or product before it is delivered to the client Also called <i>acceptance test</i> See also <i>test: unit</i> and <i>test: integration</i>	14.2
test: functional	<i>Functional</i> verification methods based on a functional model of the system to test (e.g. Finite State Machines) \diamond <i>test: structural</i>	10.5 12.3
test: functional diagnosis	Type of diagnosis techniques which aims at locating faults at functional level, without precise fault model	10.5
test: GO-NOGO	See <i>test: production</i>	
test: identification	Formal methods for test pattern generation of sequential systems without fault model	12.3
test: in situ	Test is applied to the product in its normal environment	14.6
test: integration	Test applied to sub-systems integrating elementary modules or others sub-systems	14.2
test: likelihood	See <i>likelihood test</i>	
test: localization	See <i>test: diagnosis</i>	
test: logical	Test applied to a system modeled at logical level	12.1
test: maintenance	Test applied during the maintenance operations	6.3 12.1 12.2
test: MC/DC	Structural software testing which adds the following requirement to the Condition/Decision testing method (see <i>test: condition/decision</i>): each condition in a decision must be shown to independently affect the result of the decision	13.6
test: memory	Specific testing techniques taking into account technological faults of RAM circuits: <i>checkerboard</i> , <i>marching</i> , <i>walking</i> , <i>galloping</i> or <i>ping-pong</i>	12.3
test: modified condition/decision	See <i>test: MC/DC</i>	
test: mutation	Test validation technique which consists in injecting modifications in a system in order to check whether a given test sequence detects the faults or not	13.8
test: mutation: weak	The <i>weak mutation testing</i> requires that the test sequence activates the fault introduced by the mutation, but it does not require that this sequence propagates the initial error to the outputs (as failure)	13.8
test: non-regression	Test performed after the repair of a faulty product in order to assure that no fault has been introduced by the repair operation or other changing	12.2

test: off-chip	See <i>off-chip test</i>	
test: off-line / on-line	See <i>off-line testing</i> and <i>on-line testing</i>	
test: on-chip	See <i>on-chip test</i>	
test: on-line	See <i>on-line testing</i>	
test: optimal sequence	Test sequence having a minimal length (in terms of number of test vectors)	12.3
test: parametric	Test performed on the devices to check AC and DC parameters	12.1
test: path	Software structural test technique which takes the program <i>control flow paths</i> as elements to define the test coverage	13.6
test: path sensitizing	Test pattern generation method for structural testing	13.2
test: path tracing	See <i>test: path sensitizing</i>	
test: production/ manufacturing	Technique to test each individual copy of the manufactured product to insure it was produced without defects	6.3 11.2 12.1 12.2
test: progressive	Test used to evaluate the reliability of a population of components, whose decision to stop depends on the results already obtained	7.2
test: progressive curtailed	Test used to evaluate the reliability of a population of components which is identical to the progressive test with a maximum duration constraint	7.2
test: random	Logical testing technique based on random generation of the input test vectors	12.3
test: reference list	Conventional algorithmic test procedure based on the comparison between the results produced by a tested product and a predefined list of known values stored in the tester	12.2
test: screening	Test techniques used to remove weak products according to reliability	11.2
test: signature analysis	Test method using a property on the output values of the tested product in order to evaluate its correctness	12.2
test: standard/ referent	Test method using a faultless referent product. The outputs given by the tested product and the referent product are compared	12.2
test: statement	Software structural testing technique which takes the program <i>statements</i> as structural elements to define the test coverage	13.6
test: step stress	Test used to evaluate the reliability of a population of components which provokes a progressive acceleration of the degradation mechanisms, in general by increasing the temperature (permitting an <i>accelerated test</i>)	7.2

test: structural	<p><i>Structural</i> test methods are based on a structural model (e.g. gate structure) and generally use fault model (e.g. 'stuck-at')</p> <p>See also <i>software: structural testing</i></p> <p>◇> <i>test: functional</i></p>	12.3
test: toggle	See <i>toggle test</i>	
test: unit	Test applied to elementary modules	14.2
test: validation	Validation of a test sequence, frequently by a fault grading	12.3
	See also <i>test: evaluation</i> and <i>fault grading</i>	
testability	<p>Attribute of dependability which measures the <i>easiness</i> with which a product can be tested, i.e. the <i>easiness</i> to obtain test sequences, and the <i>easiness</i> to apply these sequences</p> <p>Closely linked to the test sequence properties:</p> <ul style="list-style-type: none"> • the <i>length</i>, i.e. the number of input vectors • the <i>coverage</i> or test efficiency, i.e. the ratio of the tested fault and the total number of faults according to a given fault model <p>Testability can be evaluated on the product, by <i>controllability</i> and <i>observability</i> parameters</p> <p><i>Testability measurement</i>: methods that analyze a design and estimate the difficulty of test pattern generation as a measure of testability</p>	7.3 14.1
testability: techniques	<p>There are two are groups:</p> <ul style="list-style-type: none"> • <i>Ad hoc techniques</i>: design rules listing the structures that cause testing problems and techniques for avoiding these problems • <i>Design For testability (DFT)</i>: design techniques to increase testability 	14
tester	Any means (human or physical) involved in fault detection and diagnosis of a product by a test. Also known as <i>test equipment</i>	12.1
TMR	Triple Modular Redundancy. Basic <i>N-version</i> fault tolerant technique based on passive redundancy. Three copies (duplicate modules) of the main module are used and a voter elaborates the final output. A <i>3-version</i> also called <i>triplex</i>	18.2
toggle test	Test sequence which assures that each line of the tested component is switched to '0' and '1'	12.3
tolerable probability	See <i>risk: acceptable rate</i>	
tolerance	See <i>fault tolerance</i>	

totally self-checking system	Property of continuous on-line testing systems. A system is said to be <i>totally self-checking</i> , if it is code-preserving, self-testing and fault-secure with regard to a given fault model F <ul style="list-style-type: none"> • <i>Code-preserving</i> expresses that the fault free module preserves the output code on the observed output variables • <i>Self-testing</i> expresses that every fault of F is detectable on the tested output by at least one functional input vector • <i>Fault-secure</i> guarantees that no incorrect functional output can occur which is not immediately detectable 	16.3
traceability	Existing relationships between the elements used in a step and the elements produced by this step	9.3
triplex	See <i>TMR</i>	
trouble shooting	See <i>maintenance: troubleshooting and repair</i>	
unit test	See <i>test: unit</i>	
universe: dynamic	Set of all theoretically possible sequences of input and/or output values of a product	8.2
universe: static	Set of all theoretically possible I/O values of a product	8.2
useful life	See <i>operation</i>	
user	Entities (physical or human) interacting functionally with the product Also called <i>functional environment</i>	2.1
utilization	See <i>operation</i>	
validation	Assessment of the method used in a creation phase	6.3 9.1 10.1
VAN Bus	Vehicle Area Network: example of industrial Bus using on-line detection	18.7
vector: input	Value received or acquired by a product	8.2
vector: output	Value produced by a product	8.2
verification	Evaluation of the result of a creation phase, in order to check that it is in accordance with the requirements	6.3 9.1 10.1
version	Versions are duplicate modules that have the same specification than the original functional module. They are called <i>duplicate</i> if they have the same implementation	18.2
VHDL	See <i>HDL</i>	
vote: adaptive	Particular N-Versions technique whose erroneous versions are eliminated from the decision	18.5
voter	Module of a N-Version fault-tolerant structure which elaborates the final outputs from the outputs provided by the versions	18.2

VXI	VME eXtensions for Instrumentation. IEEE Std 1155.1992 Industry standard for test and measurement market	12.1
walking	See <i>test: memory</i>	
walkthrough	An informal review technique based on a presentation by the author and discussions between the author and the reviewer	9.4
watchdog	Mechanism to detect errors associated with deadlines which are not reached at run-time	16.3
Weibull reliability model	The <i>Weibull</i> reliability model is an interesting <i>reliability</i> model, because of its flexibility in describing a number of failure patterns	7.2
yield	Percentage of good dice (the electrical portion of the wafer that contains the electronic functions) compared to the total number of dice on the wafer. It is a statistical parameter. Yield is refined into four major yield groups: wafer processing yield, wafer probe yield, assembly yield, final test yield	12.2

References

1. E.A. Amerasekera and D.S. Campbell, *Failure Mechanisms in Semiconductor Devices*, John Wiley & Sons, 1987.
2. T. Anderson and P.A. Lee, *Fault Tolerance. Principles and Practice*, Prentice Hall International, 1981.
3. John Andrews, *Applied Fault Tree Analysis for Reliability and Risk Assessment*, Wiley Series in Quality and Reliability Engineering, Patrick D.T. O'Connor Editor, John Wiley & Sons, 2000.
4. C. Ausnit.Hood, K.A. Johnson, R.G. Pettit, and S.B. Opdahl, *Ada 95 Quality and Style*, Lecture Notes in Computer Science n° 1344, Springer-Verlag, 1997.
5. *The Evolution of Fault-Tolerant Computing*, A. Avizienis, H. Kopetz, and J.C. Laprie Editors, Springer-Verlag, 1987.
6. Michel Banatre and Peter A. Lee, *Hardware and Software Architectures for Fault Tolerance*, 311 Pages, Springer-Verlag, 1994.
7. P.H. Bardell, W.H. McAnney, and J. Savir, *Built-In Test for VLSI, Pseudo-Random Techniques*, John Willey & Sons, New York, 1987.
8. J. Barnes, *High Integrity Ada. The Spark Approach*, Addison-Wesley, 1997.
9. *Embedded Systems Applications*, C. Baron, J.C. Geffroy, and G. Motet Editors, Kluwer Academic Publishers, 1997.
10. I. Bashir, *Testing Object.Oriented Software*, Springer-Verlag, 1999.
11. L. Bening and H. Foster, *Principles of Verifiable RTL Design*, second edition, Kluwer Academic Publishers, 2001.
12. B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, 1990.
13. B. Beizer, *Black.Box Testing. Techniques for Functional Testing of Software and Systems*, John Wiley & Sons, 1995.
14. J. Bergeron, *Writing Testbenches. Functional Verification of HDL Models*, Kluwer Academic Publishers, 2000.
15. R. Billington and R.N. Allen, *Reliability Evaluation of Engineering Systems*, Plenum Press, 1982.
16. A. Birolini, *Reliability Engineering: Theory and Practice*, Springer-Verlag, 1999.
17. G. Birtwistle, and P.A. Surahmanyam, *VLSI Specification, Verification and Synthesis*, Kluwer Academic Publishers, 1988.

18. M. L. Bushnell, Vishwani and D. Agrawal, *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*, Kluwer Academic Publishers, 2000.
19. S. Chakravarty and P. Thadikaran, *Introduction to I_{DDQ} Testing*, Kluwer Academic Publishers, 1997.
20. Kwang-Ting Cheung, Vishwani and D. Agrawal, *Unified Methods for VLSI Simulation and Test Generation*, 'Series in Engineering and Computer Science: SECS73', Kluwer Academic Publishers, 1989.
21. J. M. Crichlow, *An Introduction to Distributed and Parallel Computing*, Prentice Hall, 1988.
22. R. A. DeMillo, W. Michael McCracken, R.J. Martin, and John F. Passafiume, *Software Testing and Evaluation*, The Benjamin Cummings Publishing Company, Inc., Menlo Parc, Ca. USA, 1987.
23. B. Douglass, *Real-Time UML: Developing Efficient Objects for Embedded Systems* Reading, Addison-Wesley, 1998
24. B. Douglass, *Doing Hard Time: Using Object Oriented Programming and Software Patterns in Real Time Applications* Reading, Addison-Wesley, 1999.
25. R. Drechsler, *Formal Verification of Circuits*, Kluwer Academic Publishers, 2000.
26. E. Dustin, J. Rashka, J. Paul John, and D. Mc Diarmid, *Automated Software Testing: Introduction, Management, and Performance*, Addison-Wesley, 1999.
27. N.E. Fenton, *Software Metrics. A Rigorous Approach*, Chapman and Hall, 1991.
28. M. Fowler and K. Scott, *UML Distilled: Applying the Standard Object Modeling Language* Reading, Addison-Wesley, 1997.
29. M.A. Friedman and J.M. Voas, *Software Assessment: Reliability, Safety, Testability*, Wiley, 1995.
30. T. Gilb and D. Graham, *Software Inspection*, Addison-Wesley, 1994.
31. D. Harel and M. Politi, *Modeling Reactive Systems With Statecharts: The Stateate Approach*, McGraw-Hill, 1998.
32. *Hardware Description Languages*, R.W. Hartenstein Editor, Elsevier Science Publishers, 1987.
33. *Logic Design and Simulation*, E. Höerbst Editor, Elsevier Science Publishers, 1986.
34. C.P. Hollocker, *Software Reviews and Audits Handbook*, Wiley, 1990.
35. Shi-Yu Huang, *Formal Equivalence Checking and Design Debugging*, Kluwer Academic Publishers, 1998.
36. W. Humphrey, *A Discipline For Software Engineering*, Addison-Wesley, 1995.
37. W. Humphrey, *Introduction To The Personal Software Process*, Addison-Wesley, 1997.
38. *IEEE Standard for Software Unit Testing*, IEEE Press, 1987.
39. Finn Jensen, *Component Reliability. Fundamentals, Modeling, Evaluation & Assurance*, Wiley Series in Quality and Reliability Engineering, Patrick D.T. O'Connor Editor, John Wiley & Sons, 1995.
40. Niraj K. Jha and Sandip Kundu, *Testing and Reliable Design of CMOS Circuits*, Kluwer Academic Publishers, 1990.
41. B.W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, 1989.
42. D. R. H. Jones, *Failure Analysis Case Studies: A Source Book of Case Studies Selected from the Pages of Engineering Failure Analysis 1994-1996*, Pergamon Press, 1998.
43. Cem Kaner and D. Pels, *Bad Software*, Wiley Interscience, 1998.
44. Cem Kaner, J.D. Falk and Nguyen, *Testing Computer Software*, Wiley Interscience, 1999.

45. P.K. Kapur, R.B. Garg, and S. Kumar, *Contributions to Hardware and Software Reliability Modeling*, World Scientific Publishing Company, 1999.
46. R. Kehoe A. Jarvis, *ISO 9000.3. A Tool for Software Product and Process Improvement*, Springer-Verlag, New York, 1996.
47. *Fault Tolerance: Achievement and Assessment*, M. Kersken and F. Saglietti, Editors, Strategies, Research Esprit-Project 300. Request, Vol 1, Springer-Verlag, 1992.
48. Fumihiko Kimura, *Computer-Aided Tolerancing*, Chapman & Hall, 1996.
49. Z. Kohavi, *Switching and Finite Automata Theory*, TATA McGraw Hill Publisher, 1978.
50. T. Koomen and M. Pol, *Test Process Improvement*, Kluwer Academic Publishers, 1998.
51. Way Kuo, Wei-Ting Kary Chien, and Taeho Kim, *Reliability, Yield, and Stress Burn-In: A Unified Approach for Microelectronics Systems Manufacturing and Software Development*, Kluwer Academic Publishers, 1998.
52. P. K. Lala, *Fault.Tolerant & Fault Testable Harware Design*, Prentice Hall, 1985.
53. *Dependability: basic concepts and terminology*, in five languages, J.C. Laprie Editor, IFIP WG 10.4, Springer-Verlag, 1990.
54. L. Lavagno and A. Sangiovanni-Vincentelli, *Algorithms for Synthesis and Testing of Asynchronous Circuits*, Kluwer Academic Publishers, 1993.
55. S. C. Lee, *Modern Switching Theory and Digital Design*, Prentice Hall Inc., 1978.
56. N. G. Leveson, *Safeware. System Safety and Computers*, Addison-Wesley Publishing Company, 1995.
57. *Advanced Techniques for Embedded Systems: Design and Test*, edited by Juan Carlos Lopez, Romàn Hermida, and Walter Geisselhardt, Kluwer Academic Publishers, 1998.
58. D. Luckham, *Programming with Specifications. An Introduction to ANNA, A Language for Specifying Ada Programs*, Springer-Verlag, 1990.
59. *Software Fault.Tolerance*, M.R. Lyn Editor, Wiley, 1995.
60. L. A. Macaulay, *Requirements Engineering*, Series in 'Applied Computing', Springer-Verlag, 1996.
61. B. Marick, *The Craft of Software Testing Subsystem Testing Including Object-based and Object-Oriented Testing*, Prentice Hall, 1994.
62. L. Perry Martin, *Electronic Failure Analysis Handbook*, McGraw-Hill, 1998.
63. C. Maunder and R. E. Tulloss, *The Test Access Port and Boundary Scan Architecture*, collection of several papers on this subject, IEEE Computer Society Press, Los Alamitos, Ca, USA, 1990.
64. Pinaki Mazumder and Kanad Chakraborty, *Testing and Testable Design of Random-Access Memories*, Kluwer Academic Publishers, 1996.
65. K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.
66. A. Miczo, *Digital Logic Testing and Simulation*, Harper & Row Publishers, New York, 1986.
67. C. Mitchell, V. Stavridou, *Mathematics of Dependable Systems*, Clarendon Press, 1995.
68. J. W. Moore, *Software Engineering Standards. A User's Road Map*, IEEE Computer Society, Los Alamitos, California, 1998.
69. G. Motet, A. Marpinard, and J.C. Geffroy, *Design of Dependable Ada software*, Prentice Hall, 1996.
70. G. Myers, *The Art of Software Testing*, Wiley, 1979.
71. B. Nadeau-Dostie, *Design for AT-Speed Test, Diagnosis and Measurement*, Kluwer Academic Publishers, 1999.
72. W. Nelson, *Accelerated Testing: Statistical Models, Test Plans, and Data Analyses*, Wiley Interscience, 1990.

73. P. G. Neumann, *Computer Related Risks* Addison-Wesley 1995.
74. P. D.T. O'Connor, *Practical Reliability Engineering*, 3rd edition, John Wiley & sons, 1995.
75. *Formal Methods Specification and Verification Guidebook for Software and Computer Systems*, Vol. 1 'Planning and Technology Insertion', Office of Safety and Mission Assurance, NASA, Washington DC, USA, NASA.GB.002.95, Release 1.0, July 1990.
76. A. Pages and M. Gondran, *System reliability Evaluation and Prediction in Engineering*, North Oxford Academic, 1986.
77. K. Parker, *The Boundary Scan Handbook*, second edition, Kluwer Academic Publishers, Boston, 1998.
78. L.F. Pau, *Failure Diagnostic and Performance Monitoring*, Ed. M. Dekker Inc., NY Basel, 1975.
79. W. Perry, *Effective Methods for Software Testing*, John Wiley & Sons, Inc, 1995.
80. R. M. Poston, *Automating Specification-Based Software Testing*, IEEE Computer Society Press, 1996.
81. *Fault-Tolerant Computing. Theory and Techniques*, D.K. Pradhan Editor, 2 Volumes, Prentice Hall, Englewood Cliffs, 1986.
82. *Fault-Tolerant Computer System Design*, D.K. Pradhan Editor, Prentice Hall, Englewood Cliffs, 1996.
83. P. Pukite and J. Pukite, *Modeling for Reliability Analysis: Markov Modeling for Reliability, Maintainability, Safety, and Supportability. Analysis of Complex Systems*, IEEE, 1998.
84. I.C. Pyle, *Developing Safety Systems. A Guide Using Ada*, Prentice Hall, 1991.
85. P. Rashinkar, P. Paterson, and L. Singh, *System-On-a-Chip Verification: Methodology and Techniques*, Kluwer Academic Publishers, 2000.
86. S. Robertson and J. Robertson, *Mastering the Requirements Process*, Addison-Wesley, 1999.
87. J. Rumbaugh, M. Blaha, W. Premeriani, F. Eddy and W. Lorensen, *OMT- Modeling and Object Oriented Design*, Masson & Prentice Hall, 1995.
88. D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press, Bedford Massachussets, 1982.
89. D. P. Siewiorek and R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, Third Edition, A K Peters, 1998.
90. W. R. Simpson and J. W. Sheppard, *System Test and Diagnosis*, Kluwer Academic Publishing, 1994.
91. Nozer Singpurwalla and Simon Wilson, *Statistical Methods in Software Engineering: Reliability and Risk*, Springer-Verlag, 1999.
92. D. D. Smith, *Designing Maintainable Software*, Springer-Verlag, New York, 1999.
93. I. Sommerville and P. Sawyer, *Requirements Engineering. A good practice guide*, Wiley, 1997.
94. J. T. de Sousa and P. Y.K. Cheung, *Boundary-Scan Interconnect Diagnosis*, Kluwer Academic Publishers, 2001.
95. A. M. Stavely, *Toward Zero-Defect Programming*, Addison-Wesley, 1999.
96. N. Storey, *Safety.Critical Computer Systems*, Addison-Wesley, 1996.
97. A.J. van de Goor, *Testing SemiConductor Memories. Theory & Practice*, John Wiley & Sons, 1991.
98. F. Thoen and F. Catthoor, *Modeling, Verification and Exploration of Task-Level Concurrency in Real-Time Embedded Systems*, Kluwer Academic Publishers, 1999.

99. S. A. Vanstone and P. C. van Oorschot, *An Introduction to Error Correcting Codes with Applications*, Kluwer Academic Publishers, 1989.
100. A. Villemeur, *Reliability, Availability, Maintainability and Safety Assessment: Methods & Techniques*, 2 Volumes, Wiley, 1991.
101. J. Voas and G. McGraw, *Software Fault Injection*, Wiley, 1998.
102. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Jan Vytopil Editor, Kluwer Academic Publishers, 1993.
103. E. Wallmüller, *Software Quality Assurance: A practical approach*, Prentice Hall, 1994.
104. B.A. Wichmann, *Software in Safety-Related Systems*, Wiley, 1992.
105. R. J. Wieringa, *Requirements Engineering. Framework for Understanding*, John Wiley and Sons Ltd., 1996.
106. *VLSI Testing*, T.W. Williams Editor, 'Advances in CAD for VLSI', Vol. 5, North-Holland, 1986.
107. V.N. Yarmolik, *Fault Diagnosis of Digital Circuits*, Wiley & Sons Ltd., England, 1990.
108. V.N. Yarmolik and I.V. Kachan, *Self-Testing VLSI Design*, Elsevier, Amsterdam, 1993.
109. M. Yoeli, *Formal Verification of Hardware Design*, Selection of papers, IEEE Computer Society Press Tutorial, 1990.
110. S. Zahran, *Software Process Improvement*, Addison-Wesley, 1998.

Index

A

- acceptability curve, 454
- acceptable product, 455
- acceptance test, 368, 478
- accident, 43
- activation, 71, 326
- adaptive vote, 493
- aggression, 47
- alias, 389
- alternate, 482
- arithmetic code, 419
- Arrhenius law, 148
- assertion, 246, 436
- assessment
 - dependability, 141
 - design, 212
 - error/fault model, 105
 - requirements, 204
 - reliability, 146, 479
 - risk, 452
 - safety, 453
 - testability, 363
- attribute
 - dependability, 9, 14, 154
 - module, 30, 71
 - quality, 2
- automata, 27, 270
- automatic test pattern generation (ATPG), 304, 306
- availability, 9, 154
 - instant, 154

- MDT, 154
- MUT, 155
- permanent, 154

B

- backward fault analysis, 333
- backward propagation, 327
- backward tracing, 327
- backward recovery, 476
 - acceptance test, 478
 - context restoration, 481
 - domino effect, 481
 - recovery point, 477, 479
 - recovery cache, 478
 - retry point, 477
 - rollback point, 477
 - snapshot, 479
- behavioral level, 25, 27
- behavioral model, 30
- behavioral property, 91
- benign event, 452
- Berger code, 418
- bidimensional code, 409, 415
- binary code, 404
- bit stuffing, 506
- black-box testing, 237
- boundary scan, 385
- branch test, 343
- breakdown, 6
- BSDI, 387

bug, 44
 Built-In Self-Test (BIST), 290, 366, 388
 compaction function, 290, 389
 signature, 290, 389
 signature analysis function, 290, 389
 Built-In Test (BIT), 366, 380, 387
 boundary scan (IEEE 1149-1), 385
 test bus, 385
 FIT PLA, 380
 JTAG, 385
 LSSD, 383
 scan design, 383
 full scan, 386
 partial scan, 386
 scan domain, 387
 Built-In Test Equipment (BITE), 380

C

CAN Bus, 496, 505
 catastrophic event, 452
 checker, 441, 444
 code disjoint, 445
 self-chasing checker, 445
 checksum code, 420
 client, 22
 code, 402
 code-preserving, 441
 code disjoint, 445
 codeword, 405
 cold standby, 491
 compaction, 290, 389
 compatible, 81
 compensation, 473, 486
 complete diagnosis sequence, 336
 complete distinguishing sequence, 300,
 336
 completeness, 80, 212
 compliance test, 283
 component, 30
 composition relationships, 27
 compositional hierarchy, 31
 computer aided maintenance (CAM), 287
 condition, 345
 Condition/Decision Coverage, 346
 C/DC test, 345
 confidentiality, 10, 157
 confinement, 137, 494
 conformity test, 211, 283
 consistency, 329
 constraints

 reusability, 54
 technical, 54
 technological, 54
 contamination, 73, 494
 context of the test, 432
 context of execution, 479
 contract, 23, 40
 control flow, 346
 control path, 344
 controllability, 132, 332, 364
 COTS, 54, 125
 coverage
 code, 344 408
 fault, 291
 table, 294
 test, 149, 294, 333, 363
 coverage analysis
 backward fault analysis, 333
 forward simulation, 333
 structural method, 333
 CRC, 412, 501, 503
 creation process, 22
 criticality analysis, 456
 cyclic code 412
 Cross-Interleaved Reed-Solomon
 (CIRC), 416, 502
 Reed-Solomon, 416, 502
 BCH, 415
 coding procedure, 414
 ESF, 415
 Fire, 415, 501, 508

D

dangerous event, 452
 dead-man, 435
 debugging, 281
 defect, 44
 degradation, 486
 delivered service, 19
 dependability, 9
 attribute, 9, 14
 impairments, 7, 14, 39
 means, 13
 dependability assessment, 141
 attribute, 141
 availability, 154
 forecasting value, 142
 maintainability, 152
 maintenance, 150
 exploitation value, 142

- qualitative approach, 141
 - deductive, 143
 - Fault tree Method, 168, 287
 - inductive, 143, 164
 - FMEA, 164, 287
 - FMECA, 167, 456
- quantitative approach, 141, 159
 - fault injection, 159
 - fault simulation, 159
 - Markov graph, 162
 - Monte Carlo simulation, 160
 - reliability block diagram, 160, 475
 - stochastic Petri net, 162
 - test, 144
- reliability, 145
- safety, 155
- security, 10, 157
 - confidentiality, 10, 157
 - integrity, 10, 157
- specification value, 142
- standards, 143
- testability, 149
- dependability assurance, 138
- dependability impairments, 7, 14, 39
- dependability means, 13
 - fault avoidance, 122
 - fault forecasting, 13, 142
 - fault prevention, 13, 123
 - fault removal, 13, 122, 127
 - fault tolerance, 13, 135, 138
- design, 5, 21, 24, 124, 129
 - behavioral level, 25, 27
 - electronic level, 26
 - layout, 25
 - logic level, 25
 - structural level, 25, 27
 - system, 25
 - technological level, 26, 27
- Design For Testability (DFT), 362, 365
 - Ad hoc approach, 367
 - guidelines, 367
 - boundary scan, 385
 - Built-In Self-Test, 290, 366, 388
 - Built-In Test, 366, 380, 387
 - specific design method, 377
 - Built-In Test Equipment (BITE), 380
 - combinational circuit, 377
 - Galois form, 377
 - IEEE-1149-1, JTAG, 385
 - LSSD, 383
 - Reed-Muller structure, 377
 - scan design, 383
 - software
 - exception mechanism, 374
 - instrumentation, 373
- design level, 25, 27
 - behavioral, 25, 27
 - electronic, 26
 - HDL, 25
 - logical, 25
 - mask, 26
 - structural, 25, 27
 - symbolic, 26, 27
 - system, 25
 - technological, 26, 27
 - physical, 28
 - symbolic, 26, 27
- design proof, 231
- design rule, 263
- design testing, 133
- designer, 22
- detection testing, 149, 280, 297
- development, 22
- development process, 22, 221
- device under test (DUT), 281
- diagnosis, 298, 336, 346, 489
 - adaptive sequence, 298
 - complete diagnosis sequence, 336
 - complete distinguishing sequence, 300, 336
 - diagnosis tree, 298, 336
 - distinguishing sequence, 299, 336
 - fault tree, 299
 - fixed sequence, 298
 - functional, 245
 - assertion, 246
 - post-condition, 246
 - pre-condition, 246
 - partition, 299
- diagnosis tree, 298, 338
- disastrous event, 452
- disruption, 50, 400
- distance
 - arithmetic, 420
 - Hamming, 406
- distinguishing sequence, 299, 336
- disturbance, 6, 47
- domain
 - dangerous, 459

- dynamic, 182
- functional, 180, 402
- safe, 459
- scan, 387
- static, 180, 402
- domino effect, 481
- Double-Duplex, 497
- double-rail code, 418, 443
- dreaded event, 143
- duplex, 194, 442
- duplicate, 472
- DUT, 281
- dynamic analysis, 127, 131, 237, 279
- dynamic functional domain, 182, 243

E

- easily testable system, 135
 - instrumentation, 135
 - monitoring, 135
- ECC code, 404
- EDC/ECC, 402
- Electro-Magnetic Compatibility (EMC), 273
- electronic level, 26
- emergent functionality, 82
- environment, 17
 - functional, 4, 17
 - non-functional, 4, 17
- equivalent faults
 - pattern, 297, 300
 - system, 297, 300, 338
- error, 7, 72
 - asymmetric, 95
 - burst, 401
 - confinement, 137, 494
 - contamination, 73, 494
 - diffusion, 73
 - dynamic, 73, 95
 - generic, 130
 - hard, 95
 - immediate, 75, 326
 - initial activation, 75
 - logical, 95
 - multiple, 95, 102, 401
 - order, 95, 401
 - non-logical, 95
 - overwritten, 74
 - packet, 401
 - permanent, 73, 95
 - primitive, 75, 326
 - propagation, 73, 495
 - propagation path, 73
 - single, 95, 401
 - soft, 95
 - specific, 130
 - static, 73, 95, 102
 - symmetric, 95
 - temporary, 73, 95, 102
 - transient, 73
 - unidirectional, 96
- error confinement, 137, 494
- error contamination, 73, 494
- Error Detecting and Correcting Code (EDCC), 137, 399, 402
 - anti-intrusion, 404
 - ECC, 404
 - RSA, 404
- arithmetic code, 419
 - arithmetic distance, 420
 - arithmetic treatment, 422
 - checksum code, 420
 - data storage, 422
 - logical treatment, 422
 - residual code, 420
 - modulo 9 proof, 421
 - transmission, 422
- bidimensional code, 409, 415
 - Longitudinal Redundancy Check, 415
 - Vertical Redundancy Check, 415
- binary, 404
- capacity, 408
- cardinality, 408
- codeword, 405
- cost, 408
- coverage rate, 408
- cyclic code, 412
- Cyclic Redundancy Check (CRC), 412, 501
- density, 408
- double-rail, 418, 443
- disruption, 50, 400
- disruption operator, 400
- disruption set, 401
- error correction, 402
- error detection, 402
- error model, 400, 408
- Hamming distance, 406
- Hamming theorem, 407
- linear code, 410

- control relations, 410
- Galois field, 411
- generator matrix, 411
- Hamming code, 411
- modified Hamming code, 410
- parity check bit, 412
- syndrome, 410
- systematic code, 414
- low-level coding, 404
 - HDB, 405
 - Manchester, 405, 508
 - NRZ, 404, 506, 508
- multiple parity code, 409
- m-out-of-n, 417, 443
- non-separable code, 406
- parity, 409
- power of expression, 408
- preserving, 410
- product code, 415
- redundancy, 405
- redundancy rate, 408
- separable code, 405
- single parity code, 409
- transmission
 - moment, 401
- two-rail, 418
- unidimensional code, 409
- unidirectional code, 399
 - Berger code, 418
 - m-out-of-n, 417, 443
 - two-rail code, 418, 443
- unidirectional error, 416
- error detection and correction
 - mechanisms, 136, 485
- error logging, 500, 504
- error masking, 136, 473
- error model, 91, 400, 408
 - assessment, 105
 - asymmetric, 95
 - burst error, 401
 - disruption, 400
 - disruption operator, 400
 - dynamic, 73, 95
 - error in packet, 401
 - executable code level, 104
 - logical, 95
 - multiple, 95, 102, 401
 - order, 401
 - non-logical, 95
 - permanent, 95

- single error, 95, 401
- source code level, 102
- static, 73, 95, 101
- symmetric, 95
- temporary, 73, 95, 101
- unidirectional, 96
- error recovery, 472, 485
- error typology, 91
- event tree, 169
- exception mechanism, 374, 440
- execution path, 344
- exploitation, 22
- expression means, 221
- expression tool, 221
- extraction, 231

F

- fail-fast system, 464
- fail-passive system, 464
- fail-safe system, 137, 451, 456
 - intrinsic safety, 457
 - structural redundancy, 459
- fail-silent system, 464
- failure, 3, 41
 - Byzantine, 43
 - benign, 43
 - consequence
 - external, 44
 - seriousness classes, 452
 - consistent, 42
 - crash, 43
 - disruptive, 50
 - dynamic, 42
 - external consequence, 77
 - inconsistent, 42
 - omission, 43
 - persistent, 42
 - rate, 146
 - risk, 453
 - seriousness, 43, 77, 452
 - static, 42
 - stopping, 43
 - systemic, 49
 - temporary, 42
 - timing, 42
 - value, 42
- failure mode, 42
- Failure Modes and Effect Analysis (FMEA), 164, 287
 - worksheet, 164

Failure Mode and Effects and Criticality

Analysis (FMECA), 167, 456

failure rate, 146

false alarm, 433

fault, 4, 44

accidental, 50

active, 71

classification, 63

common mode, 474, 488

component, 53

conceptual, 49

creation, 6, 48, 129

design, 49

disturbance, 6

dormant, 71

dynamic, 51

equivalent

pattern, 297, 300

system, 297, 300, 338

external, 6, 47

functional, 6, 49

hard, 95, 273, 499

hardware, 49

human-made, 49

intentional, 51

interaction, 53

intermittent, 51

internal, 6, 46

masking, 136, 192, 339, 473

module, 53

operational, 49, 58

passive, 71

permanent, 51

physical, 49

production, 49, 56

soft, 95, 273, 499

specification, 49

static, 51

structural, 69

systemic, 49

table, 294

technological, 6, 49

temporary, 51

transient, 51, 95

undetectable, 192, 326, 339

fault activation, 71, 326

initial, 75

fault avoidance, 122, 201

design, 219

validation, 221

verification, 221

expression of specifications, 209

requirement expression, 204

expression aid, 205

specification, 201, 209

validation of the method, 203

verification of the solution, 203

fault class, 63

fault collapsing, 310

fault contention, 137

fault correction, 127

fault coverage, 292

fault design, 53

fault detection, 127

fault diagnosis, 127

fault dictionary, 310

fault forecasting, 13, 142

fault grading, 306, 307, 333

deterministic, 309

probabilistic, 309

simulation, 308

statistical, 310

structural analysis, 308, 333

fault injection, 159, 308

fault isolation, 127

fault localization, 127, 297

fault logging, 441

fault masking, 339

fault model, 91

assessment, 105

bridging, 97

delay, 100

open, 97

short, 97

short-circuit, 98

software, 101

source code level, 102

stuck-off, 97, 99

stuck-on, 97, 99

stuck-open, 99

stuck-at, 96

temporal, 100

timing, 100

fault prevention, 13, 123

design, 124

design model choice, 222

design process choice, 223

design guide, 224

expression guide, 225

choice of words, 227

- lexicography, 226
- self-documenting, 227
- expression improvement, 228
- operation, 125
- production, 124
- specification, 123, 209
 - modeling process, 210
 - modeling tool, 209
- technological faults, 257
 - action on the environment, 272
 - action on the product, 261
 - design rule, 263
 - hardware technology, 258
 - software, 265
 - action on the run-time environment, 274
 - feature restriction, 267
 - hazardous feature, 274
 - implementation constraints, 275
 - language choice, 266
 - programming process
 - improvement, 269
 - programming style, 269
 - software technology, 258
- fault removal, 13, 127
 - design, 129, 229
 - formal proof, 248
 - functional diagnosis, 245
 - assertion, 246
 - post-condition, 246
 - pre-condition, 246
 - functional test, 240
 - property satisfaction, 238
 - property analysis, 239, 244
- dynamic analysis, 127
- operation, 134
- production, 133
- specification, 129, 211
 - verification, 211
 - conformity, 211
 - qualitative, 211
- static analysis, 127
- technological faults
 - off-line testing, 279
- test, 127, 149, 280, 297
- validation, 128, 203, 221
- verification, 128, 203, 221
- fault secure, 442
- fault simulation, 159, 308
 - accelerator, 310
 - concurrent, 313
 - deductive, 312
 - fault collapsing, 310
 - parallel, 312
 - serial, 311
- fault specification, 52
- fault tolerance, 13, 135, 138, 469
 - active tolerance, 485
 - adaptive vote, 493
 - backward recovery, 476
 - CAN Bus, 505
 - cold standby, 492
 - compensation technique, 473
 - confinement, 137, 494
 - Double-Duplex, 497
 - error contamination, 73, 494
 - error correction, 136
 - error detection, 136
 - error logging, 500
 - error masking, 136, 473
 - fault contention, 137
 - forward recovery, 482, 499
 - graceful degradation, 486
 - hot standby, 492
 - hot swap, 492
 - memory scrubbing, 500
 - NMR, 490
 - N-self checking, 497
 - N-Version technique, 472, 497
 - passive tolerance, 485
 - reconfiguration, 137, 486
 - recovery cache, 478
 - redundancy
 - of data, 470
 - of function, 470
 - of function & data, 470
 - retry mode, 477, 496
 - self-purging, 493
 - temporal redundancy, 471
 - TMR, 473
 - VAN Bus, 507
- fault tree, 299
- Fault Tree Method (FTM), 168, 252, 287
 - basic event, 168
- feature, 29
- final test, 368
- finite state machine, 26, 92, 233, 346
- Fire code, 415, 501, 508
- FIT PLA, 380

FMEA, 164, 287
 FMECA, 167, 456
 formal identification, 317
 formal proof, 127, 248

- deductive approach, 252
- Fault Tree Method, 252
- inductive approach, 248
- symbolic execution, 251

 forward propagation, 327
 forward simulation, 333
 forward recovery, 482, 499

- recovery block, 482
 - alternate, 482
 - version, 482
- termination mode, 483

 frequent event, 454
 functional diagnosis, 245

- assertion, 246
- post-condition, 246
- pre-condition, 246

 functional environment, 4, 17
 functional redundancy checking (FRC), 442, 536
 functional test, 237, 240, 284, 303

- likelihood, 243

G-K

generic property, 91
 guidelines, 367
 Hamming code, 411
 Hamming distance, 406
 hard faults, 95, 273, 499
 HDB(n), 405
 HDL, 25
 hierarchy, 31

- compositional, 31
- use, 31

 high reliability, 126, 145, 261, 451, 469
 hot standby, 492
 hot swap, 492
 identification, 317, 346
 impairments, 7, 14, 39
 implementation, 21
 impossible event, 454
 improbable event, 453
 in-situ maintenance, 392
 incompatibility, 81
 incompleteness, 52, 81, 123
 inconsistency, 53
 inertia, 80

input vector, 180, 288
 inspection, 215
 instrumentation, 135, 373, 440
 integration test, 368
 integrity, 10, 157
 JTAG, 385
 justification, 329
 kernel, 446

L

language, 29, 209
 latency, 75
 layout level, 25
 life cycle, 5, 21

- design, 5, 21, 24
- exploitation, 22
- implementation, 21
- manufacturing, 5, 21
- need, 21
- operation, 5, 22, 29
- phase, 21
- production, 5, 21, 28
- realization, 5, 24
- requirement, 5, 21
- specification, 5, 21, 22
- stage, 21
- step, 21
- useful life, 5, 22
- utilization, 22

 likelihood, 243, 436
 linear code, 410

- control relations, 410
- generator matrix, 411
- Hamming code, 411
- modified Hamming code, 411
- syndrome, 410
- systematic code, 414

 Linear Feedback Shift Register (LFSR), 389
 link, 30
 localization, 127, 132, 150, 485
 log file, 441
 logic level, 25
 logical test, 288, 302

- functional, 304
- structural, 304, 323

 LSSD, 383

M

maintainability, 9, 152

- Mean Time To Repair, 153
 - repair rate, 153
 - maintenance, 29, 150, 296, 392
 - CAM, 287
 - corrective, 29, 152, 296
 - curative, 152
 - evolutionary, 29, 152, 296
 - in-situ, 392
 - Mean Down Time (MDT), 154
 - Mean Up Time (MUT), 155
 - non-reparable product, 29, 127, 134, 151
 - preventive, 29, 152
 - conditional, 152, 296
 - scheduled, 152
 - systematic, 152, 296
 - remote facility, 392
 - reparable product, 29, 127, 134, 151
 - troubleshooting and repair, 150
 - maintenance test, 134, 286, 296
 - computer aided maintenance, 287
 - deep knowledge, 288
 - empirical associations, 287
 - experimental approaches, 287
 - deductive method, 287
 - inductive method, 287
 - expert system, 287
 - Fault Tree Method (FTM), 287
 - FMEA, 287
 - knowledge database, 287
 - model-based approach
 - diagnosis algorithm, 288
 - diagnosis process, 288
 - model-based approaches, 287
 - reasoning by associations, 287
 - shallow reasoning, 287
 - structure and function, 288
 - surface reasoning, 287
- major event, 452
 - Manchester code, 405, 508
 - manufacturing, 5, 21
 - Markov graph, 162
 - mask design level, 26
 - masking, 136, 199, 339, 473
 - Mean Down Time (MDT), 154
 - Mean Time Between Failures (MTBF), 147
 - Mean Time To Failure (MTTF), 147
 - Mean Time To First Failure (MTTFF), 147
 - Mean Time To Repair (MTTR), 153
 - Mean Up Time (MUT), 155
 - method, 221
 - minor event, 452
 - mission, 18
 - duration, 19
 - function, 18
 - operational lifetime, 19
 - model, 29, 209
 - behavioral, 30
 - continuous, 34
 - discrete, 34
 - structural, 30
 - structured-functional, 31
 - modeling tool, 29, 209, 221
 - feature, 29
 - Modified Condition/Decision Coverage (MC/DC), 346
 - modified Hamming code, 411
 - module, 30
 - attribute, 30
 - behavioral model, 30
 - composition, 186
 - fusion operator, 186
 - emergence operator, 187
 - hierarchy, 31
 - spare, 194, 491
 - state, 30
 - modulo 9 proof, 421
 - monitoring, 135, 434
 - Monte Carlo simulation, 160
 - m-out-of-n code, 417, 443
 - mutant, 351
 - mutation, 351
 - weak mutation testing, 354
-
- N
 - need, 21
 - netlist, 25
 - NMR, 490
 - non-functional environment, 4, 17
 - non-regression testing, 297
 - non-reparable product, 29, 127, 134
 - notation, 29
 - NRZ code, 404, 506, 508
 - N-Self Checking, 497
 - N-Versions technique, 472, 497
 - voter, 473

O

observability, 76, 132, 150, 332, 364
 off-line testing, 134, 279, 362, 366
 maintenance testing, 280
 production testing, 280
 on-line testing (OLT), 134, 137, 281, 392,
 427, 485
 continuous, 428
 discontinuous, 427
 context of the test, 432
 context saving/restoring, 432
 self-testing, 428
 operation, 5, 22, 29, 125, 134
 output vector, 180, 288

P

parallel signal analyzer (PSA), 390
 parity check, 412
 parity code, 409
 partition, 300
 pattern equivalent faults, 297, 300
 perturbation, 47
 Petri net, 25, 27
 stochastic, 162
 phase, 21
 physical property, 90
 post-condition, 246, 436
 pre-condition, 246, 436
 prime gate, 189
 probable event, 453
 process characterization & control, 264,
 283
 product, 17, 29
 product structure, 30
 component, 30
 logical link, 30
 module, 30
 sub-system, 30
 production, 5, 21, 28, 124, 133
 production testing, 133, 264, 283
 program mutation, 352
 proof, 127, 248
 propagation, 73
 backward, 327
 forward, 327
 path, 73
 property
 analysis, 239, 244
 behavioral, 91
 generic, 91

 physical, 90

 structural, 90

 property satisfaction, 238

 prototyping, 214

 pseudo-random testing, 290

Q

qualitative

 criticality analysis, 456

 dependability assessment, 143

 risk assessment, 452

 safety assessment, 456

 specification assessment, 212

 tolerance assessment, 476

quality, 1

 attribute, 2

 quality assurance test, 264

 quality control, 125, 264, 283

 destructive test, 283

 non-destructive test, 283

 quality metrics, 365

 quality of the test sequence, 363

 coverage, 363

 length, 363

 quantitative

 criticality assessment, 456

 dependability assessment, 141, 159

 risk assessment, 452

 safety assessment, 453

R

random testing, 290, 303

rare event, 453

realization, 5, 24

reconfiguration, 137, 486

reconvergent fan-out structure, 328, 334

recovery, 472, 485

 backward, 476

 block, 482, 497

 cache, 478

 forward, 482, 499

 point, 477, 479

redundancy, 11, 135, 176, 402

 active, 13, 188, 493

 code, 405

 data, 470

 function, 470

 functional, 179, 436

 domain, 402

 composition of modules

- emergence operator, 187
 - fusion operator, 186
 - dynamic domain, 182, 243
 - dynamic redundancy, 183
 - dynamic universe, 182
 - static domain, 180, 459
 - static functional redundancy, 180
 - static functional redundancy rate, 181
 - static universe, 180, 460
 - irredundant element, 188
 - passive, 136, 188, 372
 - rate, 408
 - redundant functional domain, 402
 - reusability, 55
 - semantic, 176
 - separable, 193
 - cold standby redundancy, 194
 - functional module, 193
 - hot standby redundancy, 194
 - non-separable, 193
 - off-line redundancy, 194
 - spare, 194
 - on-line redundancy, 194
 - redundant module, 193
 - software, 191
 - structural, 187, 441, 471
 - active, 188
 - gate level
 - prime gate, 189
 - hardware, 187
 - irredundant element, 188
 - passive, 188
 - software, 187
 - time, 187
 - syntactic, 176
 - temporal, 471
- Reed-Muller structure, 377
- Reed-Solomon, 416, 502
- refinement process, 33
- reliability, 9, 126, 145, 261, 451, 469
 - bathub curve, 148
 - infant mortality, 149
 - useful life, 149
 - wearout, 149
 - estimator, 146
 - evaluation, 264, 283
 - failure rate, 146
 - failure rate estimation, 148
 - accelerated test, 148
 - Arrhenius law, 148
 - Mean Time Between Failures (MTBF), 147
 - Mean Time To Failure (MTTF), 147
 - Mean time To First Failure (MTFF), 147
 - process control, 264
 - production testing, 264
 - quality control, 264
 - statistical description, 145
 - statistical mathematical tool, 145
 - reliability assurance test, 264
 - reliability block diagram, 160, 475
 - reliability evaluation, 146, 283
 - reliability mastering, 262
 - reliability model, 146
 - exponential law, 146
 - Weibull law, 147
 - reliability test, 146
 - accelerated, 148, 264
 - censured, 146
 - curtailed, 146
 - destructive/ non-destructive, 264, 283
 - progressive, 146
 - progressive curtailed, 146
 - step stress, 146
 - remote maintenance facility, 392
 - repair, 127
 - repair rate, 153
 - repairable product, 29, 127, 134
 - replica, 193, 472
 - requirement, 5, 21
 - requirement expression, 204
 - evaluation of a method, 207
 - expression aid
 - horizontal structuration, 206
 - vertical structuration, 206
 - retry mode, 477, 496
 - retry point, 475
 - reusability, 55
 - reuse, 82, 125
 - review, 127, 213, 214
 - risk, 452
 - acceptability, 454
 - acceptability curve, 454
 - acceptable product, 455
 - acceptable risk rate, 454
 - tolerable probability, 454
 - robustness, 83
 - rollback point, 477

RSA code, 404
 run-time executive, 259

S

safe state, 458
 safety, 9, 156, 433, 451, 523
 active, 458
 criticality analysis, 456
 qualitative, 456
 FMECA, 167, 456
 quantitative, 456
 dangerous domain, 459
 intrinsic, 457
 passive, 458
 safe domain, 459
 safety classes, 453
 seriousness classes, 452
 benign, 452
 catastrophic, 452
 dangerous, 452
 disastrous, 452
 major, 452
 minor, 452
 serious, 452
 significant, 452
 without effects, 452
 safety classes, 453
 extremely improbable event, 453
 extremely rare event, 453
 frequent event, 454
 impossible event, 454
 probable event, 453
 rare event, 453
 reasonably probable event, 454
 scan design, 383
 full scan, 386
 partial scan, 386
 scan domains, 387
 scenario, 213
 scrubbing, 500
 security, 10, 157
 confidentiality, 10, 157
 integrity, 10, 157
 self-checking checker, 445
 self-checking system, 441
 self-purging, 493
 self-testing, 433, 442, 465
 condition monitoring, 434
 functional redundancy, 436
 assertion, 436
 exception mechanism, 440
 instrumentation, 440
 likelihood test, 436
 post-condition, 436
 pre-condition, 436
 watchdog, 438
 observation of product operation, 434
 observation of user behavior, 435
 dead-man technique, 435
 structural redundancy, 441
 checker, 441
 code-preserving, 441
 duplex, 442
 fault-secure, 442
 self-testing, 442
 totally self-checking, 442
 self-testing system, 137, 465
 semantics, 123
 serious event, 452
 seriousness, 77, 452
 benign, 43, 77, 452
 catastrophic, 43, 78, 452
 dangerous, 78, 452
 disastrous, 78, 452
 major, 78, 452
 minor, 77, 452
 serious, 43, 78, 452
 significant, 78, 452
 seriousness class, 452
 service delivered, 19, 39
 service relationships, 27
 serviceability, 153
 severity, 77
 signature, 290
 signature analysis, 290, 389
 significant event, 452
 simulation sequence, 237
 snapshot, 479
 soft fault, 95, 499
 spare module, 194, 491
 specification, 5, 21, 22, 124, 129, 209
 contract, 23
 functional characteristics, 23
 non-functional characteristics, 23
 fault prevention, 209
 fault removal, 211
 verification, 211, 229
 stage, 21
 state, 30, 72

static analysis, 127, 129
 step, 21
 STIL, 285, 387
 stochastic Petri net, 162
 strobing, 289
 structural domain, 187
 structural fault, 69
 structural level, 25, 27
 structural property, 90
 structural software test, 340

- branch & path test, 343
- branch test, 343
- condition, 345
- condition and decision test, 345
- Condition/Decision Coverage (C/DC), 346
- control flow, 346
- control path, 344
- coverage rate, 341
- decision, 345
- execution path, 344
- Modified Condition/Decision test, 345
- mutation, 351
- weak mutation testing, 354
- path test, 344
- statement test, 342

 structural test, 305, 323
 structure, 30
 structured-functional model, 31
 stuck-at fault, 96
 stuck-on/off fault, 97, 99
 sub-system, 30
 symbolic level, 26, 27
 syndrome, 410
 system, 24, 30
 system equivalent faults, 297, 300, 338
 system level, 25

T

technical constraints, 54
 technological constraint, 54
 technological fault, 257
 technological level, 26, 27
 termination mode, 483
 test, 127, 280

- accelerated, 148, 264, 310
- acceptance test, 368, 478
- algorithmic, 303
- alpha/beta, 283

black box testing, 237
 branch, 343
 Built-In Self-Test, 290, 366, 388

- LFSR, 389

 Built-In Test (BIT), 366, 380, 387

- boundary scan, 385
- test bus, 385

 FIT PLA, 380
 IEEE 1149-1, 385
 JTAG, 385
 LSSD, 383

- scan design, 383

 burn-in, 126, 284
 BSDL, 387
 censured, 146
 compliance, 283
 condition and/or decision, 345, 346
 conformity, 283
 context, 432
 continuity, 284
 continuous, 428
 coverage, 149, 294, 333, 363
 coverage table, 294
 curtailed, 146
 design, 133
 design for testability, 365

- ad hoc technique, 365, 367
- guidelines, 367
- built-in self-test (BIST), 290, 366, 388
- built-in test (BIT), 366, 380
- specific design, 366

 destructive, 264, 283
 detection, 149, 280, 297

- fault masking, 339

 device under test (DUT), 281
 diagnosis, 149, 281, 297, 336, 346

- adaptive sequence, 299
- diagnosis tree, 299, 336
- fault tree, 299
- fixed sequence, 298
- pattern equivalent faults, 297
- system equivalent faults, 297

 discontinuous, 427
 distributed, 430
 easily testable system, 135, 362
 Embedded Core Test (IEEE P1500), 285
 exhaustive, 303
 fault coverage, 291

- fault grading, 307
 - fault injection, 308
 - fault simulation, 308
 - deterministic, 309
 - probabilistic, 309
 - statistical, 310
 - structural analysis, 307, 333
- fault localization, 127, 297
- fault table, 294
- final test, 368
- fixed diagnosis, 297
- functional, 237, 240, 284 303
- functional test sequence, 304
- generation
 - path sensitizing, 325
 - program mutation, 351
- gray box testing, 237
- IDDQ, 284
- in situ test, 392
- input sequence, 288
- integration test, 368
- likelihood, 243, 436
- localization, 149, 281
- logical, 284, 288
- maintenance, 296
- maintenance testing, 286
- Modified Condition/Decision (MC/D), 346
- non-destructive test, 264, 283
- non-regression testing, 297
- off-chip, 282, 388
- off-line, 134, 279, 362, 366
 - in situ maintenance, 392
- on-chip, 282, 388
- on-line, 134, 137, 281, 392, 427, 485
- output sequence, 288
- parametric, 283
- path, 343
- production, 133, 264, 283, 292
 - continuity, 284
 - GO-NOGO, 293
 - IDDQ, 284
 - logical, 284, 288
 - parametric, 283
 - yield, 293
- progressive, 146
- pseudo-random, 303
- RAM, 319
 - checkerboard, 319
 - galloping, 320
 - marching, 319
 - ping-pong, 320
 - walking, 319
- random, 290, 303
- reliability test, 148, 264
 - accelerated, 148, 264
 - destructive, 264
 - non-destructive, 264
- schmoo plot, 284
- screening, 265
- sequential circuit, 316
 - formal identification, 316
 - functional test sequence, 317
 - RAM, 319
- signature analysis, 290, 389
 - compaction, 290, 389
 - parallel signal analyzer (PSA), 390
- statement, 342
- step stress, 146
- STIL standard, 285, 387
- structural, 237, 304
 - algorithmic, 304
 - Automatic Test Pattern Generation, 304
 - software, 340
 - test generation, 325
- self-testing, 433
- structural test sequence, 305
 - path tracing approach, 307
- task, 431
- test point, 371
- toggle test, 303
- unit test, 368
- VXI standard, 286
 - white box testing, 237
- test application, 306
- test equipment, 281, 285, 344
 - internal, 282
 - external, 282
- test evaluation, 291, 306, 333
- test generation, 291, 302, 306, 325
 - backward propagation, 326
 - backward tracing, 326
 - consistency, 329
 - D-algorithm, 307
 - fault activation, 326
 - forward propagation, 327
 - justification, 329
 - path sensitizing, 307, 325

- primitive error, 326
- reconvergent fan-out structure, 328
- structured circuit, 332
- tracing approach, 307
- with fault model, 291
- without fault model, 291
- test pattern generation, 306, 314, 325
 - automatic, 304
 - heuristic, 315
 - optimal test sequence, 314
- test sequence, 237, 279, 288
 - adaptive, 293
 - fixed, 292
 - input, 288
 - output, 288
- test sequence quality, 149, 291, 363
 - coverage, 291, 363
 - length, 149, 291, 363
 - generation ease, 149, 291, 363
 - cost, 291
- test validation, 306
- test vector, 289
- testability, 9, 149, 362
 - controllability, 132, 332, 364
 - coverage, 149, 291, 363
 - generation ease, 149, 291, 363
 - length, 149, 291, 363
 - measurement, 363
 - observability, 132, 150, 332, 364
 - qualitative estimator, 365
 - software quality metrics, 365
 - test application, 363
 - test generation, 363
- tester, 281, 285, 288
 - signature analysis, 290
 - strobing, 289
 - with reference list, 290
 - with referent product, 290
 - pseudo-random, 290
 - random, 290
 - with standard product, 290
- toggle test, 303
- totally self-checking system, 441, 442
 - checker, 441
 - code-preserving, 441
 - fault-secure, 442
 - self-testing, 442
- troubleshooting and repair, 150
- Triple Modular Redundancy (TMR), 136, 473

- Triplex, 473
- two-rail code, 418, 443

U-Z

- unidirectional code, 416
- unit test, 368
- use hierarchy, 31
- useful life, 5, 22, 149
- user, 17, 22
- utilization, 22
- validation, 128, 203, 221
- VAN Bus, 507
- verification, 128, 203, 221
 - conformity, 211
 - design, 229
 - with specifications, 229
 - double transformation, 235
 - reverse transformation, 230
 - extraction, 231
 - top-down transformation, 236
 - property satisfaction, 238
 - simulation, 237
 - without specifications, 238
 - generic property, 238
- dynamic analysis, 237
- property, 238
- property satisfaction, 238
- qualitative
 - clarity, 212
 - completeness, 212
 - comprehension, 212
 - concision, 212
 - consistency, 212
 - non-ambiguity, 212
 - prototyping, 214
 - review, 213, 214
 - inspection, 215
 - walkthrough, 215
 - scenario, 213
 - simplicity, 212
 - traceability, 212
 - simulation, 237
- version, 193, 472, 482
- VHDL, 25
- voter, 473, 493
- walkthrough, 215
- watchdog, 438, 496
- wearout, 149
- Weibull reliability law, 147

yield, 29