ADVANCED
MICROELECTRONICS

M. Sonza Reorda
Z. Peng
M. Violante (Eds.)

# System-level Test and Validation of Hardware/Software Systems

Springer

Springer Series in

ADVANCED MICROELECTRONICS 17

Springer Series in

# ADVANCED MICROELECTRONICS

The Springer Series in Advanced Microelectronics provides systematic information on all the topics relevant for the design, processing, and manufacturing of microelectronic devices. The books, each prepared by leading researchers or engineers in their fields, cover the basic and advanced aspects of topics such as wafer processing, materials, device design, device technologies, circuit design, VLSI implementation, and subsystem technology. The series forms a bridge between physics and engineering and the volumes will appeal to practicing engineers as well as research scientists.

1 **Cellular Neural Networks**
Chaos, Complexity
and VLSI Processing
By G. Manganaro, P. Arena,
and L. Fortuna

2 **Technology of Integrated Circuits**
By D. Widmann, H. Mader,
and H. Friedrich

3 **Ferroelectric Memories**
By J.F. Scott

4 **Microwave Resonators and Filters for Wireless Communication**
Theory, Design and Application
By M. Makimoto and S. Yamashita

5 **VLSI Memory Chip Design**
By K. Itoh

6 **Smart Power ICs**
Technologies and Applications
Ed. by B. Murari, R. Bertotti,
and G.A. Vignola

7 **Noise in Semiconductor Devices**
Modeling and Simulation
By F. Bonani and G. Ghione

8 **Logic Synthesis for Asynchronous Controllers and Interfaces**
Chaos, Complexity and
VLSI Processing
By J. Cortadella, M. Kishinevsky,
A. Kondratyev, L. Lavagno,
and A. Yakovlev

9 **Low Dielectric Constant Materials for IC Applications**
Ed. by P.S. Ho, J. Leu, W.W. Lee

10 **Lock-in Thermography**
Basics and Use
for Functional Diagnostics
of Electronic Components
By O. Breitenstein
and M. Langenkamp

11 **High Frequency Bipolar Transistors**
Physics, Modelling, Applications
By M. Reisch

12 **Current Sense Amplifiers**
for Embedded SRAM
in High-Performance
System-on-a-Chip Designs
By B. Wicht

13 **Silicon Optoelectronic Integrated Circuits**
By H. Zimmerman

14 **Integrated CMOS Circuits for Optical Communications**
By M. Ingels and M. Steyaert

16 **High Dielectric Constant Materials**
VLSI MOSFET Applications
Ed. by H.R. Huff and D.C. Gilmer

Series homepage – springeronline.com

M. Sonza Reorda Z. Peng M. Violante (Eds.)

# System-level Test and Validation of Hardware/Software Systems

With 55 Figures

Springer

Professor Matteo Sonza Reorda
Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24
10129 Torino
Italy

Professor Zebo Peng
Department of Computer and Information Science
Linköping University
SE-581 83 Linköping
Sweden

Dr. Massimo Violante
Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24
10129 Torino
Italy

British Library Cataloguing in Publication Data
System-level test and validation of hardware/software
systems. — (Springer series in advanced microelectronics ; 17)
1. System design 2. Computer systems — Testing 3. Computer
software — Testing
I. Sonza Reorda, Matteo II. Peng, Zebo III. Violante, Massimo
004.2′1
ISBN 1852338997

Library of Congress Cataloging-in-Publication Data
System-level test and validation of hardware/software systems / Matteo Sonza Reorda,
Zebo Peng, Massimo Violante.
p. cm. — (Springer series in advanced microelectronics, ISSN 1437-0387 ; 17)
Includes bibliographical references and index.
ISBN 1-85233-899-7
1. Computer systems—Testing. 2. Computer programs—Testing. I. Sonza Reorda,
Matteo. II. Peng, Zebo. III. Violante, Massimo. IV. Series.
QA76.76.S64S96 2005
004.2′4—dc24 2004057798

ISBN 1-85233-899-7

Advanced Microelectronics Series ISSN 1437-0387
springeronline.com

Typesetting: Camera-ready by editors
Printed in the United States of America
69/3830-543210 Printed on acid-free paper SPIN 10983058

# Table of Contents

# Table of Figures

# List of Contributors

Bobbio, A.
Dipartimento di Informatica, Università del Piemonte Orientale
Spalto Marengo 33, 15100, Alessandria
Italy

Codetta Raiteri, D.
Dipartimento di Informatica, Università del Piemonte Orientale
Spalto Marengo 33, 15100, Alessandria
Italy

De Pierro, M.
Dipartimento di Informatica, Università di Torino
Corso Svizzera 185, 10149, Torino
Italy

Eles, P.
Dept. of Computer and Information Science, Linköping University
SE-581 83 Linköping
Sweden

Franceschinis, G.
Dipartimento di Informatica, Università del Piemonte Orientale
Spalto Marengo 33, 15100, Alessandria
Italy

Fummi, F.
Dipartimento di Informatica, Università di Verona
Via dell'Artigliere 8, 37129, Verona
Italy

Goloubeva, O.
Dipartimento di Automatica e Informatica, Politecnico di Torino
Corso Duca degli Abruzzi 24, 10129, Torino
Italy

Harris, I. G.
Dept. of Information and Computer Science, University of California Irvine
442 Computer Science Building, Irvine, CA 92697-3425
USA

Jervan, G.
Dept. of Computer and Information Science, Linköping University
SE-581 83 Linköping
Sweden

Peng, Z.
Dept. of Computer and Information Science, Linköping University
SE-581 83 Linköping
Sweden

Pravadelli, G.
Dipartimento di Informatica, Università di Verona
Via dell'Artigliere 8, 37129, Verona
Italy

Sánchez, E.
Dipartimento di Automatica e Informatica, Politecnico di Torino
Corso Duca degli Abruzzi 24, 10129, Torino
Italy

Sonza Reorda, M.
Dipartimento di Automatica e Informatica, Politecnico di Torino
Corso Duca degli Abruzzi 24, 10129, Torino
Italy

Squillero, G.
Dipartimento di Automatica e Informatica, Politecnico di Torino
Corso Duca degli Abruzzi 24, 10129, Torino
Italy

Teixeira, J. P.
Instituto Superior Técnico, Technical University of Lisbon
Rua Alves Redol 9, 3º,1000-029, Lisboa
Portugal

Ubar, R.
Tallinn Technical University
Ehitajate tee 5, EE0026, Tallinn
Estonia

Violante, M.
Dipartimento di Automatica e Informatica, Politecnico di Torino
Corso Duca degli Abruzzi 24, 10129, Torino
Italy

# 1 Introduction

Z. Peng, M. Sonza Reorda, M. Violante

Linköping University, Linköping, Sweden
Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy

When looking at the evolution of the electronic circuit's design process in the past, one can clearly see a trend going from lower to higher abstraction levels: designers have been focusing their main efforts on designs at higher and higher levels of abstraction over the years. In this way designers can manage the always-growing size and complexity of circuits and systems, and leave the handling of details at the lower levels to computer-aided design tools.

This trend has been particularly visible in recent years, due to the emergence of new manufacturing technologies that allow the integration of entire systems on single chips (systems-on-chip, or SOCs). The possibility of manufacturing complex SOCs has paved the road for unprecedented levels of integration, and the SOC technology has entailed many challenges, in particular to the designers.

To cope with the design challenge, tools and techniques addressing designs at the system level of abstraction have been developed and used in the industry. Many SOC implementation details can thus be neglected during the design process; designers can, therefore, focus their efforts on the definition of the system's behavior that best fits with the user needs while analyzing the cost/benefit tradeoff of several different architectures.

Besides the design practice, the advent of SOCs is also reshaping the way validation and test activities are handled, which have begun to migrate from the register-transfer or gate levels towards the system level. Unfortunately, the current status is that system-level design tools do not support test and validation, although some research efforts have already addressed these issues.

The introduction of system-level design tools has indeed unified the process of specification of the different modules composing SOCs. When their behavior has been finalized, designers can decide which modules will be implemented in software and which ones in hardware, depending on the design's goals and constraints. In this scenario, provided that suitable techniques are available to deal with system-level specifications, designers have the possibility of addressing validation and test activities of both hardware and software almost independently from the final system implementation. Is this reasonable? Are the current obstacles only connected to the lack of deep understanding of what system-level descriptions can provide, or is there hope for really starting the test and validation process from system-level descriptions? The European Union funded a research project named COTEST (*Testability Support in a Co-design Environment*) with the explicit goal of providing a first answer to these questions.

This book gathers some of the most important results produced in the framework of COTEST, as well as an in-depth overview of the state of the art about system-level validation and testing techniques written by several experts in the field. It covers several important issues of system-level test and validation, including bug/defects modeling, stimuli generation for validation/test purposes (including timing errors) and design for testability.

The second chapter of this book, authored by J.P. Teixeira, aims firstly at correlating high-level fault models with lower level ones, and at investigating the possibility of developing test patterns and strategies without detailed information about the structure of the final system. Based on the results, new techniques for developing test patterns are also proposed.

The third chapter, by F. Fummi and G. Pravadelli, assumes that a functional fault model has been selected, and that some input test pattern targeting it must be generated out of a high-level description. Symbolic techniques are combined with random ones and integrated in a high-level environment for test and validation.

The chapter by O. Goloubeva, M. Sonza Reorda, and M. Violante explores the feasibility of an approach where test vectors are generated out of a system-level description, before the partitioning between software and hardware modules is performed. Therefore, test generation is completely independent of the implementation of each module, and resulting vectors can be exploited either to test the hardware components, or to validate software code.

The fifth chapter, by G. Jervan *et al.*, proposes some techniques that combine high-level information with low-level information in order to produce optimal test vectors. A hierarchical approach is exploited, which leads to high fault-coverage figures with minimal computational efforts. The method is based on a flexible representation of the system, based on decision diagrams.

The sixth chapter, authored by E. Sánchez *et al.*, describes the very specific problem of testing processor cores starting from high-level descriptions, and reports an analysis of several high-level testing metrics. The authors also show how an evolutionary-computation paradigm can be applied successfully to the purpose of testing complex processors.

The seventh chapter, by I.G. Harris, focuses on the very important issue of tackling with concurrency and timing problems. The author overviews the main inter-process synchronization mechanisms and proposes a metric that can measure the effectiveness of a test suite in detecting possible errors in their implementation.

The subsequent chapter, by G. Jervan *et al*, covers the last frontier in system-level testing, *i.e.*, the issue of taking care of test constraints and introducing design for testability already during the system-level design process. In particular, it discusses a technique to combine deterministic and random patterns in a hybrid BIST approach and how to select appropriate hybrid BIST architecture for a given design.

The final chapter by A. Bobbio *et al.* gives an insight about formal techniques that can be exploited for analyzing complex systems. In this chapter most of the details of system components are neglected, and only a few parameters are used to characterize them. The authors show how these parameters can be exploited effectively for validating the dependability property of the systems studied.

It is our belief that this book represents a good overview of existing system-level test and validation techniques, and we hope that its publication will boost new research activities in the field.

## Acknowledgments

We would like to thank the European Union, which supported most of the work this book is based on through the COTEST project, which has been carried out under the framework of the IST Information Society Technologies FET Open Programme, VI.1.1 Future and Emerging Technologies.

# 2 Modeling Permanent Faults

J. P. Teixeira

IST / INESC-ID, Lisboa, Portugal

## 2.1 Abstract

Test and validation of the hardware part of a hardware/software (HW/SW) system is a complex problem. Design for Testability (DfT), basically introduced at structural level, became mandatory to constrain design quality and costs. However, as product complexity increases, the test process (including test planning, DfT and test preparation) needs to be concurrently carried out with the design process, as early as possible during the top-down phase, starting from system-level descriptions. How can we, prior to the structural synthesis of the physical design, estimate and improve system testability, as well as perform test generation? The answer to this question starts with high-level modeling of permanent faults and its correlation with low-level defect modeling. Hence, this chapter addresses this problem, and presents some valuable solutions to guide high-quality test generation, based on high-level modeling of permanent faults.

## 2.2 Introduction

Test and validation of the hardware part of a hardware/software (HW/SW) system is a complex problem. Assuming that the design process leads to a given HW/SW partition, and to a target hardware architecture, with a high-level behavioral description, *design validation* becomes mandatory. Is the high-level description accurately representing all the customer's functionality? Are customer's specifications met? Simulation is used extensively to verify the functional correctness of hardware designs. *Functional tests* are used for this purpose. However, unlike formal verification, functional simulation is always incomplete. Hence, coverage metrics have been devised to ascertain the extent to which a given functional test *covers* system functionality [10]. Within the design validation process, often diagnosis and *design debug* are required.

System-level behavioral descriptions may use software-based techniques, such as Object-Oriented Modeling techniques and languages such as Unified Modeling Language [2,8], or more hardware-oriented techniques and Hardware Design Languages (HDLs), such as SystemC [23], VHDL or Verilog. Nevertheless, sooner or later a platform level for the hardware design is reached – Register-Transfer Level (RTL). The synthesis process usually starts from this level, transforming a *behavioral* into a *structural* description, mapped into a target library of primitive elements, which can be implemented (and interconnected) in a semiconductor device.

Hence, a hierarchical set of description levels – from system down to physical (layout) level – is built, the design being progressively detailed, verified, detailed again and checked for correctness. This *top-down* approach is followed by a *bottom-up* verification phase, in which the compliance of the physical design to the desired functionality and specifications is ascertained. Such verification is first carried out in the design environment, by simulation, and later on through prototype validation, by physical test, after the first silicon is manufactured. Understandably, RTL is a *kernel level* in the design process, from behavioral into structural implementation, from high-level into low-level design.

When structural synthesis is performed, often testability is poor. Hence, DfT, basically introduced for digital systems at structural level, becomes mandatory to boost design quality and constrain costs. Structural reconfiguration, by means of well-known techniques, such as scan and Built-In Self-Test (BIST) [4,33], is introduced. Automatic Test Pattern Generation (ATPG) is performed (when deterministic vectors are required), to define a high-quality *structural test.* Logic-level fault models, like the classic single Line Stuck-At (LSA) fault model [9], are used to run fault simulation. The quality of the derived test pattern is usually measured by the *Fault Coverage* (FC) metrics, *i.e.*, the percentage of listed faults detected by the test pattern.

Performing testability analysis and improvement only at structural level is, however, too late and too expensive. Decisions at earlier design crossroads, at higher levels of abstraction, should have been made to ease the test process and increase the final product's quality. In fact, as product complexity increases, the test process (including test planning, DfT and test preparation) needs to be concurrently carried out with the design process, as early as possible during the top-down phase. How can we, prior to the structural synthesis of the physical design, estimate and improve system testability, as well as perform useful test generation?

Test generation is guided by *test objectives* (Figure 2.1). At higher levels of abstraction, functional tests are required for design validation. In order to constrain the costs of functional tests, usually some DfT is introduced. At this level, DfT is inserted to ease system partitioning in coherent and loosely connected modules, and to allow their accessibility. *Product validation* requires a more thorough test. In fact, the physical structure integrity of each manufactured copy of the design needs to be checked. This requires a *structural test.* Certain applications, namely highly dependable or safety-critical ones, require *lifetime test*, *i.e.*, the ability to test the component or system in the field. Lifetime test also requires a structural test.

Functional tests are usually of limited interest for later production or lifetime test. Could high-level test generation be useful to build functional test patterns that could be reused for such purposes? Usually, hardware Design & Test (D&T) engineers are very skeptical about test preparation and DfT prior to structural synthesis. After all, how can we generate a test to uncover faults at a structure yet to be known? Can we perform test generation that may be *independent* from the structural implementation? In fact, how useful can a functional test be to achieve 100% (structural) fault coverage?

Although such skepticism has its arguments (and plain experimental data), it is relevant to consider the cost issue, as far as test preparation is concerned. Design *reuse* [18] has been identified as a key concept to fight the battle of design productivity, and to lower design costs. If *test reuse* could also be feasible, it would certainly be a significant added value to enhance test productivity and lower its costs. Hence, pushing the test process up-stream, to higher levels of abstraction (and earlier design phases), has been a driving force in current test research efforts. This chapter focuses on analyzing current trends on modeling PFs using higher levels of system description, to enable design debug, high-level DfT and to guide high-level TPG in such a way that test reuse for production or lifetime testing of manufactured hardware parts can be performed.

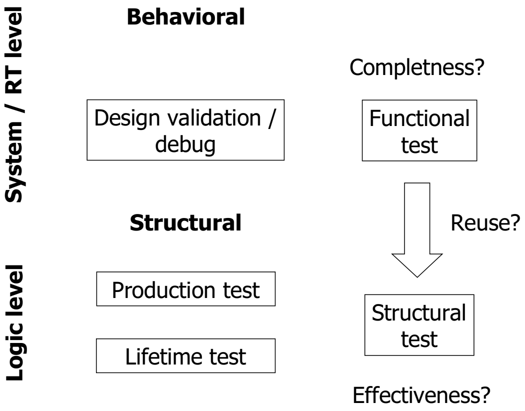


**Figure 2.1.** Test objectives

Modeling PFs to achieve this goal must address two questions:

- What fault models should be used, to generate test patterns, which may be useful for design validation and for structural testing?
- How can we define Quality Metrics (QMs) to ascertain the test development quality, at RTL, prior to physical synthesis?

This chapter is organized as follows. In Section 2.3, the underlying concepts and their definitions are provided. Section 2.4 identifies the relevant high-level QMs to ascertain the usefulness of a given test solution, focusing on the key role of Test Effectiveness (TE) (defined in Section 2.3). In Section 2.5, a review of the system-level and RTL fault models available is performed, in order to identify their usefulness and limitations. Finally, Section 2.6 summarizes the main conclusions of this chapter.

## 2.3 Definitions

In the context of this work, the following definitions are assumed:

- **Time-Invariant System –** a HW/SW system which, under specified environmental conditions, *always* provides the same response *Y*(*X*,*t*) when activated by a specific sequence of input stimuli, *X*, applied at its Primary Inputs (PIs), whatever the time instant, $t_o$, at which the system starts being stimulated.
- **Defect** – any physical imperfection, modifying circuit topology or parameters. Defects may be induced during system manufacturing, or during product lifetime. ***Critical defects***, when present, modify the system's structure or circuit topology.
- **Disturbance** – any defect or environmental interaction with the system (*e.g.*, a Single Event Upset, SEU), which may cause abnormal system behavior.
- **Fault** – any abnormal system behavior, caused by a disturbance. Faults represent the *impact* of disturbances on system behavior. ***Catastrophic faults*** are the result of critical defects. The manifestation of disturbances as faults may occur or not, depending on test stimuli, and may induce abnormal system behavior:
    1. Locally (inside a definable damaged module), or at the circuit's observable outputs (here referred to as Primary Outputs (POs));
    2. Within the circuit's response time frame (clock cycle), or later on, during subsequent clock cycles.
- **Error** – any abnormal system behavior, observable at the system's POs. Errors allow fault detection (and, thus, defects detection).
- **Permanent Fault (PF)** – any abnormal system behavior that transforms the fault-free time-invariant system into a new time-invariant system, the faulty one. Non-permanent faults affect system behavior only at certain time intervals, and are referred as ***intermittent faults***.

Typically, environmental interactions with the system, such as SEUs caused by alpha particles, induce intermittent faults. Hence, the only disturbances considered in this chapter are physical defects, causing permanent PFs.

- **Fault Model –** an abstract representation of a given fault, at a specified level of abstraction, typically, the same level at which the system or module is described. Fault models accurately describe the fault's *valued characteristics* within the model's *domain of validity*. A useful glossary of fault models is provided in [4].
- **Test Pattern –** a unique sequence of ***test vectors*** (input digital words) to be applied to the system's PIs.
- **TE** – the ability of a given test pattern to uncover disturbances (*i.e.*, to induce errors, in their presence) [40].

## 2.4 High-level Quality Metrics

*Quality* is not measured as an absolute value. It has been defined as "the totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implicit needs" [20]. Hence, the quality of a product or of its design is a measure of the fulfillment of a given set of *valued characteristics*. Nevertheless, different observers value different characteristics. For instance, company shareholders value product profitability. Product users value the ability of the product to perform its required functionality, within a given performance range, in a stated environment, and for a given period of time. However, a trade-off between quality and cost is always present. Hence, product quality is bounded by how much the customer is willing to pay.

Quality evaluation and improvement require the definition of the valued characteristics, and of QMs. As far as electronic-based systems are concerned, QM, presented in the literature, are basically separated into software metrics [19,25] and hardware metrics [5,12,14,16,21,24,36,37,38,39]. *Software metrics* aim at measuring software design and code quality. Traditional metrics, associated with functional decomposition and a data analysis design approach, deal with design structure and/or data structure independently. Valued characteristics are architectural complexity, understandability/usability, reusability and testability/maintenance. *Hardware metrics* also aim at measuring architectural complexity, provided that performance specifications are met. However, product quality requirements make testability a mandatory valued characteristic. Although these QMs come from two traditionally separated communities, hardware design is performed using a *software model* of the system under development. Prior to manufacturing, all development is based on the simulation of such a software model, at different abstraction levels. Therefore, it is understandable that effort is made to reuse SW quality indicators in the HW domain. For instance, in the SW domain, module (object) *cohesion* and *coupling* metrics are accepted as relevant quality indicators. The image of this, in the HW domain, deals with modularity, interconnection complexity among modules and data traffic among them.

In new product development, it becomes relevant to design, product, process and test quality. The assessment of design productivity, time-to-market and cost effectiveness are considered in *design quality*.

*Product quality* is perceived from the customer's point of view: the customer expects the shipment of 100% defect-free products, performing the desired functionality. A usual metric for product quality is the Defect Level (DL), or escape rate. The DL is defined as the percentage of defective parts that are considered as good by manufacturing test and thus marketed as good parts [32]. This metric is extremely difficult to estimate; in fact, if the manufacturer could discriminate *all* defective parts, zero-defects shipments could be made, and customer satisfaction boosted. Field rejects usually provide feedback on the product's DL, and they have a strong impact on the customer/supplier relationship and trust, as well as heavy costs. DL is computed in Defects Per Million (DPM), or parts per million (ppm). Quality products have specifications in the order to 50–10 DPM or even less. DL depends strongly on test quality.

*Process quality* is associated with the ability of the semiconductor manufacturing process to fabricate good parts. Process quality is measured by the process *yield* (*Y*), and evaluated by the percentage of manufactured good parts. This metric's accuracy also depends on test quality, as the measured yield is obtained through the results of production tests and its ability to discriminate between good and defective parts.

*Test quality*, from a hardware manufacturing point of view, basically has to do with the ability to discriminate good from defective parts. This deals with production and lifetime testing. However, if validation test is considered, the functional test quality needs to assess the comprehensiveness of verification coverage [10]. As the test process has a strong impact on new product development, several characteristics of the test process are valued, and several metrics defined (Table 2.1). The necessary condition for test quality is TE. TE has been defined as the ability of a given test pattern to uncover physical defects [40]. This characteristic is a challenging one, as new materials, new semiconductor, Deep Sub-Micron (DSM) technologies, and new, unknown defects emerge. Additionally, system complexity leads to the situation of billions of possible defect locations, especially spot defects. Moreover, the accessible I/O terminals are few, compared with the number of internal structural nodes. The basic assumption of production test of digital systems is the modeling of manufacturing defects as logical faults. However, how well do these logical faults represent the impact of defects? This issue is delicate, and will be addressed in Section 2.5.

In order to increase TE and decrease the costs of the test process, system reconfiguration (through DfT techniques) is routinely used. But it has costs, as some *test functionality* migrates into the system's modules (*e.g.*, Intellectual Property (IP) cores in Systems-on-Chip). Hence, additional, valued characteristics need to be considered as sufficient conditions for test quality: Test Overhead (TO), Test Length (TL) and Test Power (TP).

*TO* estimates the additional real estate in silicon required to implement test functionality (*e.g.*, 2% Si area overhead), additional pin-count (*e.g.*, four mandatory pins for BoundaryScan Test [4]) and degradation on the system's performance (lower speed of operation).

*TL* is the number of test vectors necessary to include in the test pattern to reach acceptable levels of TE. TL is a crucial parameter; it impacts (1) test preparation costs (namely, the costs of fault simulation and ATPG), (2) test application costs, in the manufacturing line, and (3) manufacturing throughput.

Finally, *TP* addresses a growing concern: the average and peak power consumption required to perform the test sessions [13]. In some applications, the corresponding energy, *E*, is also relevant. In fact, scrutinizing all the system's structure may require large circuit activity, and thus an amount of power which may greatly exceed the power consumption needed (and specified) for normal operation. This may severely restrict test application time.

In this chapter, we focus on TE for two reasons. First, it is a *necessary* condition: without TE, the test effort is useless. Second, in order to develop test patterns at system or RTL that may be reused at structural level, this characteristic must be taken into account. Accurate values of TE, TO, TL and TP can only be computed at structural level.

**Table 2.1.** Test quality assessment

| Valued characteristics | Quality metrics |
|---|---|
| TE | FC, Defects Coverage |
| TL, test application time | $N$, # of test vectors |
| TP | $P_{\text{AVE}}$, $E$ |
| TO: | Test overhead: |
| • Silicon area | • % area overhead |
| • Pin count | • # additional pins |
| • Speed degradation | • % speed degradation |

TE is usually measured, at structural level, through the FC metrics. Given a digital circuit with a structural description, *C*, a test pattern, *T*, and a set of *n* listed faults (typically, single LSA faults), assumed equally probable, if *T* is able to uncover *m* out of *n* faults, FC = *n/m.* Ideally, the designer wants FC = 100% of detectable faults. Nevertheless, does FC = 100% of listed faults guarantee the detection of *all* physical defects? In order to answer such a question, a more accurate metric has been defined, *Defects Coverage* (DC) [32].

Defects occur in the physical semiconductor device. Hence, defect modeling should be carried out at this low level of abstraction. However, simulation costs for complex systems become prohibitive at this level. Hence, at minimum, defect modeling should be performed at logic level, as LSA faults have originally been used at this level of abstraction. For a manufacturing technology, a set of *defect classes* is assumed (and verified, during yield ramp-up). A *defects statistics* is also built internally, as yield killers are being removed, and production yield climbs to cost-effective levels. A list of likely defects can then be built, extracted from the layout, *e.g.* using the Inductive Fault Analysis (IFA) approach [31]. However, the listed defects are not equally probable. In fact, their likelihood of occurrence is an additional parameter that must be considered. For a given system with a structural description, *C*, a test pattern, *T*, and a set of *N* listed defects, DC is computed from

$$\text{DC} = \frac{\sum_{j=1}^{N_{\text{d}}} w_j}{\sum_{i=1}^{N} w_i} \tag{2.1}$$

where $w_j$ is the fault weight, $w_j = -\ln(1 - p_j)$ and $p_j$ is the probability of occurrence of fault *j* [32]. Hence, TE is weighted by the likelihood of occurrence of the $N_d$ defects uncovered by test pattern *T*.

## 2.5 System and Register-transfer-level Fault Models for Permanent Faults

Test and validation aim at identifying deviations from a specified functionality. Likely deviations are modeled as PFs. PFs, as defined in Section 2, are caused by defects. If design validation is to be considered, code errors may be viewed as "PFs", in the sense that they deviate the system's behavior from its correct behavior. For software metrics, we assume such an extended meaning of "PF".

The proposed QMs can be classified as software and hardware metrics. *Software metrics* aim at measuring software design and code quality. In the software domain, a Control Flow Graph (CFG) can represent a program's control structure. Input stimuli, applied to the CFG, allow identifying the statements activated by the stimuli (test vectors). The *line coverage* metric computes the number of times each instruction is activated by the test pattern. *Branch coverage* evaluates the number of times each branch of the CFG is activated. *Path coverage* computes the number of times every path in the CFG is exercised by the test pattern. High TE for software testing can be obtained with 100% path coverage. However, for complex programs, the number of paths in the CFG grows exponentially, thus becoming prohibitive. An additional technique, used in the software domain, is *mutation testing*. Mutation analysis is a fault-based approach whose basic idea is to show that particular faults cannot exist in the software by designing test patterns to detect these faults. This method was first proposed in 1979 [7]. Its objective is to find test cases that cause faulty versions of the program, called *mutants*, containing one fault each, to fail. For instance, condition IF ($J < I$) THEN may be assumed to be erroneously typed (in the program) as IF ($J > I$) THEN. To create mutants, a set of *mutation operators* is used to represent the set of faults considered. An operator is a fault injection rule that is applied to a program. When applied, this rule performs a simple, unique and syntactically correct change into the considered statement.

Mutation testing has been used successfully in software testing, in design debug, and has been proposed as a testing technique for hardware systems, described using HDL [1,21]. It can prove to be useful for hardware design validation. Nevertheless, it may lead to limited structural fault coverage, dependent on the mutant operators selected. Here, the major issue is how far a mutant injection drives system performance away from the performance of the original system. In fact, if its injection drives system functionality far away, fault detection is easy. This fact can be computed by the percentage of the input space (possible input vectors) that leads to a system response different from the mutant-free system. Usual mutants, like the one mentioned above, may lead to easily detectable faults. However, the art of a hardware production test is the ability to uncover *difficult*-to-detect faults. Even random test vectors detect most easily detectable faults. Deterministic ATPG is required to uncover difficult-to-detect faults.

When extending software testing to hardware testing, two major differences need to be taken into account:

- As the number of observable outputs (POs) at a hardware module, or core, is very limited compared with the number of internal nodes, hardware test provides much less data from POs than software does through the reading of memory contents.
- As hardware testing aims at screening the manufactured physical structure of each design copy, fault models derived at high level (system, or RTL) cannot represent *all* logical faults that may be induced by defects in the physical structure.

The first difference makes *observability* an important characteristic, which needs to be valued in hardware testing. The second difference requires that all high-level fault models, and their usage, must be analyzed with respect to their representativeness of structural faults. In the following sections, three approaches (and corresponding metrics) are reviewed, one for design verification (Observability-based Code Coverage Metrics (OCCOM)), and two for high level TPG (Validation Vector Grade (VVG) and Implicit Functionality, Multiple Branch (IFMB)). But, first let us review the basic RTL fault models proposed in the literature for hardware testing.

Several RTL fault models [1,11,15,26,27,28,29,34,42] (see Table 2.2) and QMs [5,12,14,16,21,24,36,37,39] have been proposed. Controllability and observability are key valued characteristics, as well as *branch coverage,* as a measure of the thoroughness by which the functional control and data paths are activated and, thus, considered in the functional test. In some cases, their effectiveness in covering single LSA faults on the circuit's structural description has been ascertained for some design examples. However, this does not guarantee their effectiveness to uncover physical defects. This is the reason why more sophisticated gate-level fault models (*e.g.*, the bias voting model for bridging defects) and detection techniques (current and delay) have been developed [4]. In [28] a functional test, applied as a complement to an LSA test set, has been reported to increase the DC of an ALU module embedded in a public domain PIC controller. The work reported in [27, 28, 29] also addresses RTL fault modeling of PFs that drive ATPG at RTL, producing functional tests that may lead to high DC, at logic level.

Fault models in variables and constants are present in all RTL fault lists and are the natural extension from the structural level LSA fault model. *Single-bit* RTL faults (in variables and constants) are assumed, like classic *single* LSA faults, at logic level. The exception is the fault model based in a software test tool [1], where constants and variables are replaced using a mutant testing strategy. Such a fault model leads to a significant increase in the size of the RTL fault list, without ensuring that it contributes significantly to the increase of TE. Remember, however, that considering single LSA faults at variables and constants, defined in the RTL behavioral description, does not include single LSA faults at many structural logic lines, generated after logic synthesis.

Two groups of RTL faults for logical and arithmetic operators can be considered: replacement of operators [1, 42] and functional testing of building blocks [15]. Replacement of operators can lead to huge fault lists, with no significant coverage gains. Exhaustive functional testing of all the building blocks of every operator is an implementation-dependent approach, which leads to good LSA fault coverage as shown in [15]. Nevertheless, it may not add significantly to an increase in TE.

**Table 2.2.** RTL fault model classes

| RTL fault model classes | [1] | [42] | [26] | [11] | [15] | [27] |
|---|---|---|---|---|---|---|
| LSA type | | X | X | X | X | X |
| Logic / arithmetic operations | X | X | | | X | |
| Constant / variable switch | X | | | | | |
| Null statements | X | X | X | | | X |
| IF / ELSE | X | X | X | X | X | X |
| CASE | X | X | X | X | X | X |
| FOR | X | | X | X | | X |

The Null statement fault consists in not executing a single statement in the description. This is a useful fault model because fault detection requires three conditions: (1) the statement is reached, (2) the statement execution causes a value change of some state variable or register, and (3) this change is observable at the POs. Conditions (1) and (2) are controllability conditions, while (3) is an observability condition. Nevertheless, fault redundancy occurs when both condition (IF, CASE) faults and Null statement faults are included in the RTL fault list. In fact, Null statement faults within conditions are already considered in the condition fault model. Moreover, a Null statement full list is prohibitively large; hence, only sampled faults (out of conditions) of this class are usually considered.

The IF/ELSE fault model consists of forcing an IF condition to be stuck-at true or stuck-at false. More sophisticated IF/ELSE fault models [1] also generate the dead-IF and dead-ELSE statements. CASE fault models include the control variable being stuck to each of the possible enumerated values (CASE stuck-at), disabling one value (CASE dead-value), or disabling the overall CASE execution. In [26], a FOR fault model is proposed, by adding and subtracting one from the extreme values of the cycle. Condition RTL faults have proved to be very relevant in achieving high LSA and defects coverage [29].

The RTL fault models depicted in Table 2.1, when used for test pattern generation and TE evaluation, usually consider *single* detection. Therefore, when *one* test vector is able to detect a fault, this fault is dropped (in fault simulation, and ATPG). However, *n-detection* can significantly increase DC. Thus, as we will see in Section 2.5.3, the authors in [27, 28, 29] impose *n*-detection for condition RTL faults. RTL fault model usefulness is generally limited by the fault list size and by the fault simulator mechanisms available for fault injection.

### 2.5.1 Observability-based Code Coverage Metrics

Fallah *et al.* [10] proposed OCCOM for functional verification of complex digital system hardware design. An analogy with fault simulation for validating the quality of production test is used. The goal is to provide hardware system designers an HDL coverage metrics to allow them to assess the comprehensiveness of their simulation vector set. Moreover, the metrics may be used as a diagnostic aid, for design debug, or for improvement of the functional test pattern under analysis.

In this approach, the HDL system description is viewed as a structural interconnection of modules. The modules can be built out of combinational logic and registers. The combinational logic can correspond to Boolean operators (*e.g.*, NAND, NOR, EXOR) or arithmetic operators (*e.g.*, +, >). Using an event-driven simulator, controllability metrics (for a given test pattern) can easily be computed. A key feature addressed by these authors is the computation of an observability metric. They define a single tag model. A *tag* at a code location represents the possibility that an incorrect value was computed at that location [10]. For the *single* tag model, only one tag is identified and propagated at a time. The goal, given a test pattern (functional test) and an HDL system description, is to determine whether (or not) tags injected at each location are propagated to the system's PO. The percentage of propagated tags is defined as *code coverage* under the proposed metrics. A two-phase approach is used to compute OCCOM: (1) first, the HDL description is modified, eventually by the addition of new variables and statements, and the modified HDL descriptions are simulated, using a standard HDL simulator; (2) tags (associated with logic gates, arithmetic operators and conditions) are then injected and propagated, using a flow graph extracted from the HDL system description.

Results, using several algorithms and processors, implemented in Verilog, are presented in [10]. OCCOM is compared with line coverage. It is shown that the additional information provided by observability data can guide designers in the development of truly high-quality test patterns (for design validation). As it is not a stated goal of the OCCOM approach, no effort is made to analyse a possible correlation between high OCCOM values (obtained with a given test pattern) and eventual high values of the structural FC of logical faults, such as single LSA faults. Hence, no evaluation of OCCOM, with respect to TE, is available. As a consequence, no guidelines for RTL ATPG are provided.

### 2.5.2 Validation Vector Grade

Thaker *et al.* [34,35] also started by looking at the validation problem, improving code coverage analysis (from the software domain) by adding the concepts of observability and an arithmetic fault library [34]. In this initial paper, these authors first explore the relationship between RTL code coverage and LSA gate-level fault coverage, FC. As a result, they proposed a new coverage metric, *i.e.*, VVG, both for design validation and for early testability analysis at RTL. Stuck-at fault models for every RTL variable are used. They analyzed the correlation between RTL code coverage and logic level FC, and this provided input on the definition of RTL fault lists which may lead to good correlation between VVG and FC values. Under the VVG approach, an RTL variable is reported covered only if it can be controlled from a PI and observed at a PO using a technique similar to gate-level fault grading [22]. Correlation between VVG and FC is reported within a 4% error margin.

The work has evolved in [35], where the RTL fault modeling technique has been explicitly derived to predict, at RTL, the LSA FC at structural gate-level. The authors show, using a timing controller and other examples, that a *stratified* RTL fault coverage provides a good estimate (0.6% error, in the first example) of the gate-level LSA FC. The key idea is that the selected RTL faults can, to some extent, be used as a *representative* subset of the gate-level LSA fault universe. In order to be a representative sample of the collapsed, gate-level fault list, selected RTL faults should have a distribution of detection probabilities similar to that for collapsed gate faults. The *detection probability* is defined as the probability of detecting a fault by a randomly selected set of test vectors. Hence, given a test pattern comprising a sequence of *n* test vectors, and a given fault is detected *k* times during fault simulation (without fault dropping), its detection probability is given as *k/n.* If the RTL fault list is built according to characteristics defined in [35], the two fault lists exhibit similar detection probability distributions and the corresponding RTL and gate-level fault coverages are expected to track each other closely within statistical error bounds.

The authors acknowledge that not all gate-level faults can be represented at RTL, since such a system description does not contain structural information, which is dependent on the synthesis tool, and running options. Therefore, the main objective of the proposed approach is that the defined RTL fault models and fault injection algorithm are developed such as the RTL fault list of each system module becomes a representative sample of the corresponding collapsed logic-level fault list. The proposed RTL model for permanent faults has the following attributes:

- language operators (which map onto Boolean components, at gate level) are assumed to be fault free;
- variables are assigned with LSA0 and LSA1 faults;
- it is a *single* fault model, *i.e.*, only one RTL fault is injected at a time;
- for each module, its RTL fault list contains input and fan-out faults;
- RTL variables used more than once in executable statements or the instantiations of lower level modules of the design hierarchy are considered to have fan-out. Module RTL *input* faults have a one-to-one equivalence to module *input* gate-level faults. Fan-out faults of variables *inside* a module, at RTL, represent a subset of the fan-out faults of the correspondent gate-level structure.

The approach considers two steps. First, an RTL fault model and injection algorithm is developed for single, stand-alone modules. Second, a *stratified sampling technique* [4] is proposed for a system built of individual modules [35]. Hence, the concept of *weighted* RTL LSA FC, according to module complexity, has been introduced to increase the matching between this coverage and gate-level LSA FC. Faults in one module are classified in a subset (or *stratum),* according to relative module complexity. Faults in each module are assigned a given weight. The stratified RTL FC is computed taking into account the contribution of all *strata* and serves, thus, as an accurate estimation of the gate-level, LSA FC of the system. No effort is made to relate FC with DC.

### 2.5.3 Implicit Functionality, Multiple Branch

The majority of approaches proposed in the literature evaluate TE through the FC value for single LSA faults. The reasons for this are relevant. First, this fault model is independent on the actual defect mechanisms and location. In fact, it assumes that the impact of all physical defects on system behavior can be represented as a local bit error (LSA the complementary value driven by the logic), which may (or not) be propagated to an observable PO. Second, LSA-based fault simulation algorithms and ATPG tools are very efficient, and have been improved for decades. Third, the identification of defect mechanisms, defect statistics and layout data is usually not available to system designers, unless they work closely with a silicon foundry. Moreover, as manufacturing technology moves from one node to the following one [17], emerging defects are being identified, as a moving target. Although the 100% LSA FC does not guarantee 100% DC, it provides a good estimate (and initial guess) of TE for PFs. Later on in the design flow, TE evaluation may be improved using defects data.

Modeling all defect mechanisms, at all possible (and likely) locations at the layout level, is a prohibitive task. As stated, fault models at logic level have been derived to describe their impact on logic behavior. For instance, a bridging defect between two logic nodes ($X$,$Y$) may, for a given local vector, correspond to a *conditional* LSA0 at node $X$, provided that, in the fault-free circuit, the local vector sets (1) $X$ = 1 and (2) $Y$ (the dominant node, for this vector) = 0. Hence, in the fault-free circuit, ($X$,$Y$) = (1,0) and in the faulty one, ($X$,$Y$) = (0,0). Using a test pattern generated to detect single LSA faults, if the $X$ stuck-at 0 fault is detected many times, thus ensuring the propagation of this fault to an observable output, there is a good probability that, at least in one case, the local vector forces (in the fault-free circuit) $X$ = 1 and $Y$ = 0. Hence, the test pattern, generated for LSA fault detection, has the ability of covering this *non-target fault*. Therefore, the quality of the derived test pattern, *i.e.*, its ability to lead to high DC (DL) values, depends on its ability to uncover non-target faults. This is true not only for different fault models or classes, at the same abstraction level, but also at different abstraction levels. This principle is verified in the VVG approach, were RTL fault lists are built to become a representative sample of logic-level fault lists. If a given test pattern (eventually, generated to uncover RTL faults) is applied to the structural description of the same system, it will also uncover the corresponding gate-level LSA faults.

The IFMB approach uses this principle, in its two characteristics:

- accurate defect modeling is required for DC evaluation [30], but *not* for ATPG: in fact, test patterns generated to detect a set of target faults are able to uncover many non-target faults, defined at the same or at different abstraction levels [41];

- in order to enhance the likelihood of a test pattern to uncover non-target faults (which may include defects of emerging technologies, yet to be fully characterized), *n-detection* of the target faults (especially those hard to detect) should be considered [3].

Hence, as shown in Figure 2.2, consider a given RTL behavioral description leading to two possible structures, A and B, with different gate-level fault lists ("Logic LSA"), but with the same RTL fault list ("RTL faults"). Thaker *et al.* built the RTL fault list [35] as a representative sample of the gate-level LSA fault list, but they do not comment on how sensitive the matching between RTL and gate-level FC is to structural implementation. In the IFMB approach, the authors look for a correlation between *n-detection* of RTL faults and single defects detection (as measured by DC).

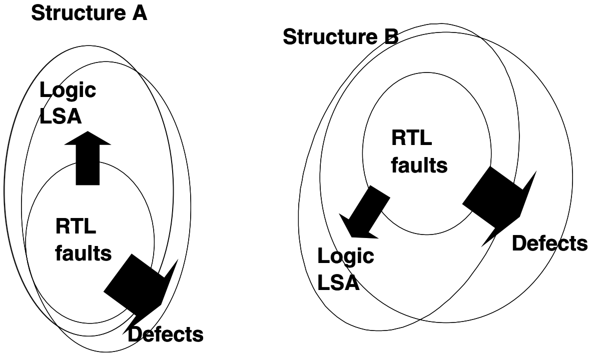


**Figure 2.2.** Representativeness of RTL, gate-level LSA faults and physical defects

An additional aspect, not considered this far, is the *types of test pattern* to be used. An external test uses a *deterministic test*, to shorten TL and thus test application time and Automatic Test Equipment (ATE) resources. On the contrary, BIST strategies [4, 33] require extensive use of pseudo-random (PR) on-chip TPG, to reduce TO costs. A deterministic test usually leads to *single* detection of hard faults, in order to compact the test set. Moreover, a deterministic test targets the coverage of specific, modeled fault classes, like LSA; in fact, it is a *fault-biased* test generation process. If, however, there is no certainty on which defect (and, thus, fault) classes will finally be likely during manufacturing, a PR test, being unbiased, could prove to be rewarding to some extent. In other words, if *PR pattern-resistant faults* were uncovered by some determinism, built in the PR test stimuli, then a PR-based test could prove to be very effective to detect physical defects. The BIST strategy can also be rewarding to perform an at-speed test, at nominal clock frequency, allowing one to uncover dynamic faults, which become more relevant in DSM technologies. BIST is also very appropriate to lifetime testing, where no sophisticated ATE is available.

The IFMB coverage metrics are defined as follows. Consider a digital system, characterized by an RTL behavioral description *D*. For a given test pattern, $T=\{T_1, T_2, \ldots T_N\}$, the IFMB coverage metrics are defined as

$$\mathrm{IFMB} = \frac{N_{\mathrm{LSA}}}{N_{\mathrm{F}}}\mathrm{FC}_{\mathrm{LSA}} + \frac{N_{\mathrm{IF}}}{N_{\mathrm{F}}}\mathrm{FC}_{\mathrm{IF}} + \frac{N_{\mathrm{MB}}}{N_{\mathrm{F}}}\mathrm{FC}_{\mathrm{MB}}(n) \tag{2.2}$$

where $N_{\mathrm{LSA}}$, $N_{\mathrm{IF}}$, and $N_{\mathrm{MB}}$ represent the number of RTL LSA, implicit functional and conditional constructs faults respectively. The total number of listed RTL faults, $N_{\mathrm{F}}$ , is $N_{\mathrm{F}} = N_{\mathrm{LSA}} + N_{\mathrm{IF}} + N_{\mathrm{MB}}$. Hence, three RTL fault classes are considered. Each individual FC defined in IFMB is evaluated as the percentage of faults in each class, single ($\mathrm{FC}_{\mathrm{LSA}}$, $\mathrm{FC}_{\mathrm{IF}}$) or *n*-detected ($\mathrm{FC}_{\mathrm{MB}}$), by test pattern *T*.

IFMB is, thus, computed as the weighted sum of three contributions: (1) *single* RTL LSA FC ($\mathrm{FC}_{\mathrm{LSA}}$), (2) *single* Implicit Functionality (IF) fault coverage ($\mathrm{FC}_{\mathrm{IF}}$) and (3) *multiple* conditional constructs faults Multiple Branch (MB) coverage ($\mathrm{FC}_{\mathrm{MB}}(n)$). The multiplicity of branch coverage, *n*, can be user defined. The first RTL fault class has been considered in the work of previous authors (Table 2.2, [1,11,15,26,27,42]). The two additional RTL fault classes and the introduction of *n*-detection associated with MB are proposed in [29], and incorporated in the IFMB quality metric. Here, the concept of *weighted* RTL FC is used in a different way than in [35] by Thaker *et al.* In fact, in the IFMB approach, the RTL fault list is partitioned in three classes, all listed faults are assumed equally probable, and the weighting is performed taking into account the relative incidence of each fault class in the overall fault list. The inclusion of faults that fully evaluate an operator's implicit functionality also differs from the model proposed in [34], where faults are sampled from a mixed structural-RTL operator description.

One of the shortcomings of the classic LSA RTL fault class is that it only considers the input, internal and output variables *explicit* in the RTL behavioral description. However, the structural implementation of a given functionality usually produces *implicit variables,* associated with the normal breakdown of the functionality. In order to increase the correlation between IFMB and DC, the authors add, for some *key implicit variables,* identified at RTL, LSA faults at each bit of them. The usefulness of IF modeling is demonstrated in [29] for relational and arithmetic (adder) operators. For instance, the IF RTL fault model for *adders and subtractors* includes the LSA fault model at each bit of the operands, of the result *and at the implicit carry* bits. This requires that such *implicit variable* is inserted in the RTL code, which needs to be expanded to allow fault injection.

Conditional constructs can be represented in a graph where a *node* represents one conditional construct, a *branch* connects unconditional execution and a *path* is a connected set of branches. Branch coverage is an important goal for a functional test. However, the usual testability metrics do not ensure more than *single branch activation and observation*, which is not enough to achieve acceptable DC values. Hence, a *multi-branch* RTL fault model for conditional constructs is proposed in [29]. It inhibits and forces the execution of each CASE possibility and forces each

IF/ELSE condition to both possibilities. Two testability metrics are defined for each branch:

**Branch Controllability:**

$$\mathrm{CO}_i = \mathrm{CO}_i(b_i) = \begin{cases} n_{\mathrm{a}i} / n, n_{\mathrm{a}i} < n \\ 1, n_{\mathrm{a}i} \geq n \end{cases} \tag{2.3}$$

where $n_{\mathrm{a}i}$ is the number of activations of branch $b_i$, and

**Branch Detectability:**

$$\mathrm{DO}_i = \mathrm{DO}_i(b_i) = \begin{cases} n_{\mathrm{d}i} / n, n_{\mathrm{d}i} < n \\ 1, n_{\mathrm{d}i} \geq n \end{cases} \tag{2.4}$$

where $n_{\mathrm{d}i}$ is the number of times the non-execution of branch $b_i$ was detected. The contribution of conditional constructs to the IFMB global metric is then defined as

$$\mathrm{FC}_{\mathrm{MB}}(n) = \sum_i^{\mathrm{NF}} \frac{\mathrm{DO}_i(n)}{\mathrm{NF}} = \sum_i^{\mathrm{NF}} \frac{n_{\mathrm{d}i}}{n.\mathrm{NF}} \tag{2.5}$$

A typical profile of branch detectability is shown in Figure 2.3. The simulation tool performs RTL fault simulation with, as an example, 5000 random vectors, and performs fault dropping when a given RTL fault is detected 30 times. As can be seen in Figure 2.3, a subset of RTL faults is not detected, or detected a few times. These are the target faults for deterministic generation. After this, the shaded area (in gray) will be significantly smaller, and both IFMB and DC metrics will approach 100%.

The authors show in [29] that a correlation of 95% between IFMB and DC is obtained with *n*-detection of $n = 5$, for several ITC'99 benchmarks [6]. The proposed RTL fault models are implemented in a proprietary mixed-level simulation environment, `VeriDOS` [30]. Low-cost RTL simulation allows the identification of parts of the RTL code difficult to exercise (referred as *dark corners*). Dark corners contain hard functionality. The identification of the parts of the functionality difficult to test is then used to identify partially specified test vectors, referred to as *masks*, which may significantly increase IFMB. The premise here is that, despite logic minimization performed during logic synthesis (which makes it more difficult to map the RTL description into the logic description), such hard functionality will be somehow mapped in structural parts that will be hard to scrutinize. Hence, *multiple* RTL fault detection increases the probability of detection of single defects on the synthesized structure.

For each mask *i,* $m_i$ fixed positional bits in possible input words are identified and assigned. Typically, $m_i$ is much smaller than the total number of PIs. Unassigned bits are filled with PR Boolean values, *e.g.* generated by an on-chip *Linear Feedback Shift Register* (LFSR). Performing mask generation for all hard RTL faults drives the "illumination" of all dark corners, and the generation of a consistent set of $n_{cv}$ constrained vectors, $X_i$ $(i = 1,2,..., n_{cv})$, or masks. Typically, $n_{cv} < 20$.

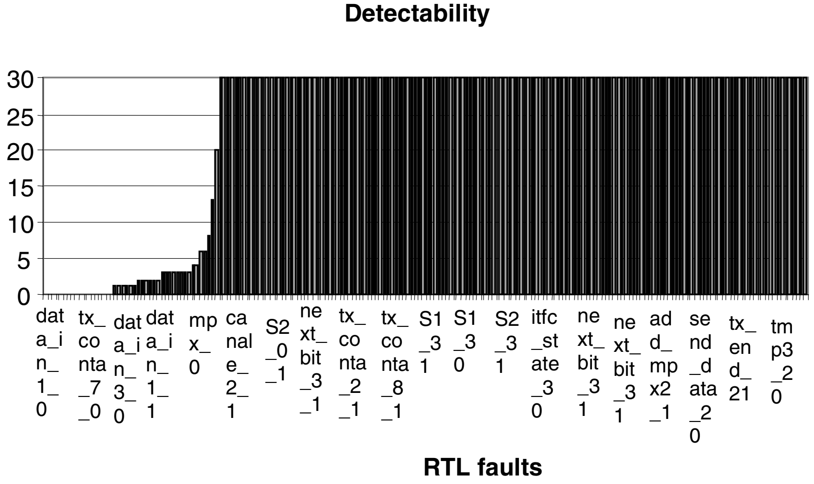


**Figure 2.3.** Typical RTL fault detectability profile

The proposed methodology is thus suitable for a BIST implementation, since random test generation needs only to be complemented with a few masks, constraining a few $m_i$ positional bits. Hence, only a loosely deterministic self-test generation is required. As relative by short test sequences can ensure high TE, low-energy BIST may thus be achieved. A typical example is shown in Figure 2.4. The authors also refer to the IFMB approach as *masked-based BIST,* or *m-BIST.*

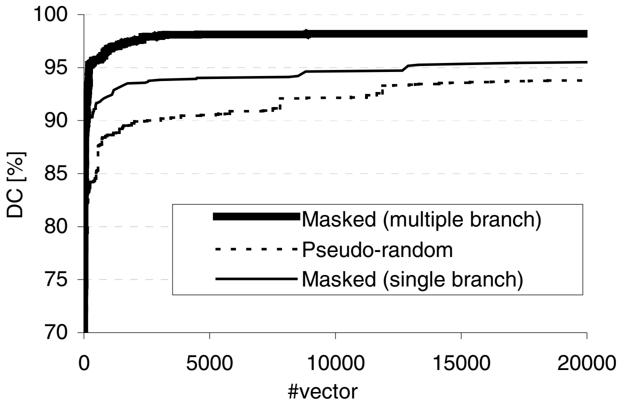


**Figure 2.4.** Typical DC improvement, using IFMB

## 2.6 Conclusions

Modeling PFs is a critical factor of success in design validation and product testing. It depends strongly on test objectives. At system level and RTL, design validation is the main objective. Thus, *functional tests* need to be derived to assess how thoroughly system functionality is verified. For production and lifetime testing, a *structural test* pattern is required. Test reuse makes it very rewarding to develop techniques that allow functional test generation that may be reused (and/or improved) for product test.

As electronic design is performed, for integrated systems, using a software model of the system to be manufactured, effort has been made to reuse software testing techniques in hardware testing. However, two key aspects need to be taken into account: (1) observability in hardware is much more limited than in software, and (2) high-level fault models cannot model all logical faults that may occur in the physical structure.

Design, product, process and test quality need to be ascertained. This chapter focused on *test quality* assessment and improvement. Four QMs have been identified, as valued characteristics: TE, TO, TL and TP. TE is a mandatory characteristic; hence, TE has been in the spotlight. TE, considered at system level or RTL, must take into account the correlation between high-level FC metrics, and low-level DC.

Several RTL fault models are available in the literature. However, the TE obtained with a given functional test is usually ascertained only with its ability to uncover single LSA faults on the logic structure, generated by logic synthesis tools. Three approaches have been highlighted. First, OCCOM has been identified as a valuable observability-based code coverage metric. It is promising for design validation, design debug and functional test improvement. Nevertheless, it does not target test pattern generation for later reuse in production or lifetime testing.

The stratified RTL fault coverage approach (and the corresponding VVG metric) is a valuable approach to derive high-quality tests for production testing. The underlying idea is that, for each module, the assumed RTL faults become a *representative* sample of the gate-level LSA faults of its corresponding structure. According to model complexity, faults associated with each module become a *stratum.* Stratified sampling techniques are then used to compute the overall RTL FC. RTL faults are derived that lead to a very accurate matching between RTL FC, and gate-level LSA FC. However, no data are available on possible DC effectiveness.

Finally, the IFMB approach (and metric) is reviewed. This approach derives RTL fault models for explicit and implicit variables in the RTL code, and imposes *n-detection* of conditional RTL faults to significantly enhance the correlation between IFMB and DC. This approach also accommodates different defect models, by introducing the concept of partially defined test vectors (referred as masks) and using PR filling of the non-assigned bits. The key idea is to achieve full branch coverage, by activating hard functionality, identified at RTL. Test length is relaxed, compared with a deterministic test, but TE is significantly increased. The approach is especially suitable for BIST solutions. As no approach can be totally implementation independent, during the bottom-up verification of the design, test pattern quality improvement can be performed, but requires a marginal effort.

## Acknowledgments

I want to thank Professor Marcelino Santos and Professor Isabel Teixeira for all the collaborative work and useful discussions on high-level fault modeling and test preparation.

## References

[1] Al Hayek G, Robach C (1996) From specification validation to hardware testing: a unified method. In: Proc. of IEEE Int. Test Conference, 885-893

[2] Baldini A, Benso A, Prinetto P, Mo S, Taddei A (2001) Towards a unified test process: from UML to end-of-line functional test. In: Proc. Int. Test Conference, 600-608

[3] Benware B, Schuermyer Ch, Ranganathan S, Madge R, Krishnamurthy P, Tamarapalli N, Tsai H-H, Rajski J (2003) Impact of multiple-detect test patterns on product quality. In: Proc. of IEEE Int. Test Conference, 1031-1040

[4] Bushnel ML, Agrawal VD (2000) Essentials of electronic testing for digital memory and mixed-signal VLSI circuits. Kluwer Academic Publishers

[5] Chickermane V, Lee J, Patel JH (1994) Addressing design for testability at the architectural level. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 13(7): 920-934

[6] CMUDSP benchmark (I-99-5, ITC 99 5]), http://www.ece.cmu.edu/~lowpower/benchmarks.html

[7] DeMillo R, Lipton R, Sayward F (1978) Hints on test data selection: help for the practicing programmer. IEEE Computer, 11: 34-41

[8] Dias OP, Teixeira IC, Teixeira JP (1999) Metrics and criteria for quality assessment of testable HW/SW system architectures. Journal of Electronic Testing: Theory and Application, 14(1/2): 149-158

[9] Eldred RD (1959) Routines based on symbolic logic statements. Journal of ACM, 6(1): 33-36

[10] Fallah F, Devadas S, Keutzer K (2001) OCCOM: efficient computation of observability-based code coverage metrics for functional verification. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 20(8): 1003-1015

[11] Ferrandi F, Fummi F, Sciuto D (1998) Implicit test generation for behavioral VHDL models. In: Proc. Int. Test Conference, 587-596

[12] Gentil MH, Crestani D, El Rhalibi A, Durant C (1994) A new testability measure: description and evaluation. In: Proc. of IEEE VLSI Test Symposium, 421-426

[13] Girard P (2002) Survey of low-power testing of VLSI circuits. IEEE Design & Test of Computers, 19(3): 82-92

[14] Gu X, Kuchcinski K, Peng Z (1994) Testability analysis and improvement from VHDL behavioral specifications, In: Proc. EuroDAC, 644-649

[15] Hayne JR, Johnson BW (1999) Behavioral fault modeling in a VHDL synthesis environment. In: Proc. of IEEE Int. Test Conference, 333-340

[16] Hsing C, Saab DG (1993) A novel behavioral testability measure. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 12(12): 1960-1970

[17] http://public.itrs.net/Files/2003ITRS/Home2003.htm

[18] Keating M, Bricaud P (1999) Reuse methodology manual for system-on-a-chip designs. 2nd Edition, Kluwer Academic Publishers.

[19] Kitchenham B, Pfleeger SL (1996) Software quality: the elusive target. IEEE Software, January 1996: 12-21.

[20] ISO 8402 International Standards (1996) "Quality – vocabulary", International Organization for Standardization (ISO)

[21] Le Traon Y, Robach C (1995) From hardware to software testability. In: Proc of IEEE Int. Test Conference, 710-719

[22] Mao W, Gulati R (1996) Improving gate-level fault coverage by RTL fault grading. In: Proc. of IEEE Int. Test Conference, 150-159

[23] Muller W, Rosenstiel W, Ruf J (2003) SystemC: methodologies and applications. Kluwer Academic Publishers.

[24] Papachristou C, Carletta J (1995) Test synthesis in the behavioral domain. In: Proc. of IEEE Int. Test Conference, 693-702

[25] Pressman RS (1997) Software engineering: a practitioner's approach. McGraw-Hill

[26] Riesgo Alcaide T (1996) Modelado de fallos y estimación de los processos de validación funcional de circuitos digitales descritos en VHDL sintetizable. PhD Thesis, Escuela Téc. Sup. Ing. Industriales, U.P. Madrid

[27] Santos MB, Gonçalves FM, Teixeira IC, Teixeira JP (2002) RTL design validation, DFT and test pattern generation for high defects coverage. Journal of Electronic Testing: Theory and Application, 18(1): 177-185

[28] Santos MB, Gonçalves FM, Teixeira IC, Teixeira JP (2001) RTL-based functional test generation for high defects coverage in digital systems. Journal of Electronic Testing: Theory and Application, 17(3 and 4): 311-319

[29] Santos MB, Gonçalves FM, Teixeira IC, Teixeira JP (2001) Implicit functionality and multiple branch coverage (IFMB): a testability metric for RT-level. In: Proc. of IEEE Int. Test Conference, 377-385

[30] Santos MB, Teixeira JP (1999) Defect-oriented mixed-level fault simulation of digital systems-on-a-chip using HDL. In: Proc. of the Design Automation and Test in Europe Conference, 549-553

[31] Shen JP, Maly W, Ferguson FJ (1985) Inductive fault analysis of MOS integrated circuits. IEEE Design & Test of Computers, 2(6): 13-26

[32] Sousa JJT, Gonçalves FM, Teixeira JP, Marzocca C, Corsi F, Williams TW (1996) Defect level evaluation in an IC design environment. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 15(10): 1286-1293

[33] Stroud ChE (2002) A designer’s guide to built-in self test. Kluwer Academic Publishers

[34] Thaker PA, Agrawal VD, Zaghloul ME (1999) Validation vector grade (VVG): a new coverage metric for validation and test. In: Proc. IEEE VLSI Test Symposium, 182-188

[35] Thaker PA, Agrawal VD, Zaghloul ME (2003) A test evaluation technique for VLSI circuits using register-transfer level fault modeling. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 22(8): 1104-1113

[36] Thearling K, Abraham JA (1989) An easily functional level testability measure. In: Proc. of IEEE Int. Test Conference, 381-390

[37] Vahid M, Orailoglu A (1995) Testability metrics for synthesis of self-testable designs and effective test plans. In: Proc. IEEE VLSI Test Symposium, 170-175

[38] Vranken HPE, Wittman MF, van Wuijtswinkel RC (1996) Design for testability in hardware-software systems. IEEE Design & Test, 13(3): 79-87

[39] Wallack JR, Dandapani R (1994) Coverage metrics for functional tests. In: Proc. of IEEE VLSI Test Symposium, 176-181

[40] Wang LC, Mercer R, Williams TW (1995) On the decline of testing efficiency as fault coverage approaches 100%. In: Proc. IEEE VLSI Test Symposium, 74-83

[41] Wang LC, Mercer R (1996) Using target faults to detect non-target defects. In: Proc. of IEEE Int. Test Conference, 629-638

[42] Ward PC, Armstrong JR (1990) Behavioral fault simulation in VHDL. In: Proc. 27th ACM/IEEE Design Automation Conference, 587-593

# 3 Test Generation: A Symbolic Approach

F. Fummi, G. Pravadelli

Università di Verona, Dipartimento di Informatica, Verona, Italy

## 3.1 Abstract

Automatic test pattern generation (ATPG), based on random methods, is widely applied in different flavors to detect functional faults. In particular, genetic algorithms, which can be considered part of the random-based category, work quite well for a quick exploration of the test patterns space, achieving good fault coverage in a short time. However, a certain number of faults are hard to detect for random-based approaches. On the contrary, such faults can be easy to detect for other strategies, as for example symbolic test pattern generation. Nevertheless, faults that are easy to detect for random-based ATPG may be, indeed, hard to detect for symbolic techniques. Addressing this class of faults, a symbolic ATPG can require longer execution time than a random one, and in some cases it can be unable to generate test sequences for some faults. This work shows how a hybrid approach, where symbolic ATPG is applied after a random-based ATPG, may represent a valuable solution to achieve a very high fault coverage keeping low the execution time. The testing methodology is implemented in a highly flexible functional verification framework that is based on a high-level fault model and a test generation strategy applicable for both random-based and symbolic-based approaches. The applicability and the efficiency of the functional testing framework presented have been confirmed by the benchmarks analyzed.

## 3.2 Introduction

By following the classical literature on testing [1], stuck-at faults in gate-level circuits can be partitioned into two classes:

- **Easy-To-Detect (ETD) faults.** ETD faults can be detected by selecting one test sequence from a large set of test sets. Generally, they are close to the Primary Inputs (PIs) and/or primary outputs and their activation and propagation can be simply performed.
- **Hard-To-Detect (HTD) faults.** HTD faults can be detected by using very few (sometimes only one) sequences. They are responsible for the consumption of the largest part of CPU time during the test generation. We can thus say that

HTD faults require a large amount of time to be detected while ETD faults are detected in a small CPU time. The same concepts can be moved to functional testing by considering high-level faults instead of stuck-at faults. However, the concept of HTD and ETD functional faults depends on the Automatic Test Pattern Generation (ATPG) algorithm used to detect them. In fact, a fault can be ETD for some functional ATPG approaches, but HTD for some other techniques. This consideration is enforced by comparing the characteristics of random-based ATPGs with respect to symbolic ATPGs and *vice versa*.

Let us examine functional testing from the point of view of an ATPG based on a genetic algorithm (GA), which is considered a very effective approach for random-based test pattern generation. Note that analogous considerations can be argued with respect to other random-based approaches too. A GA-based ATPG identifies a test pattern for a target fault by performing crossover operations, which correspond to a move to a different portion of the solution space, to arrive at an input configuration close to the solution. Each portion of this space is then more accurately explored by executing mutation operations, until the actual test pattern is identified. We can say that faults belong to the HTD class if a large number of crossover operations must be performed before arriving so close to the solution that mutation operations can identify it. In other words, faults that can be detected by using few test vectors are HTD for GA, since many crossover operations must be performed on average to get close to such few test vectors. HTD faults thus have the same meaning for GA-based ATPG and for traditional gate-level ATPG.

On the contrary, let us consider functional testing from the point of view of Binary Decision Diagrams (BDDs) (see Section 3.3), *i.e.*, a symbolic approach. A typical BDD-based test pattern generator consists of comparing the BDD representing the fault-free design with the BDD representing the faulty design for each modeled fault. The fault-free BDD and the faulty BDD are incrementally generated by analyzing the Control Flow Graph (CFG) extracted from the description of the Device Under Test (DUT) [12]. A different BDD is created for every node of the CFG and for every bit of signals, ports and variables involved in the function represented by the node. All these partial BDDs are collected by a BDD manager that creates a Compact Representation (CR) of all the BDD nodes involved. The size of this compact representation, in terms of collected BDD nodes, is proportional to the number of differences between every fault-free partial BDD and the corresponding faulty partial BDD. If a fault can be detected by using a large set of test patterns, then there are many different faulty partial BDDs, which are different from the fault-free BDD, for each faulty behavior. In this case, the CR generated by the BDD manager is composed of many BDD nodes, since it represents the fault-free behavior and all faulty behaviors. Therefore, test generation becomes computationally expensive, since the construction time of the CR is proportional to its size. Moreover, in this case a state-space explosion problem may arise, making the generation of the CR not feasible. Thus, HTD faults, for a BDD-based ATPG, are those faults that can be detected by using a large set of test patterns; that is, those faults that are ETD for a GA-based ATPG.

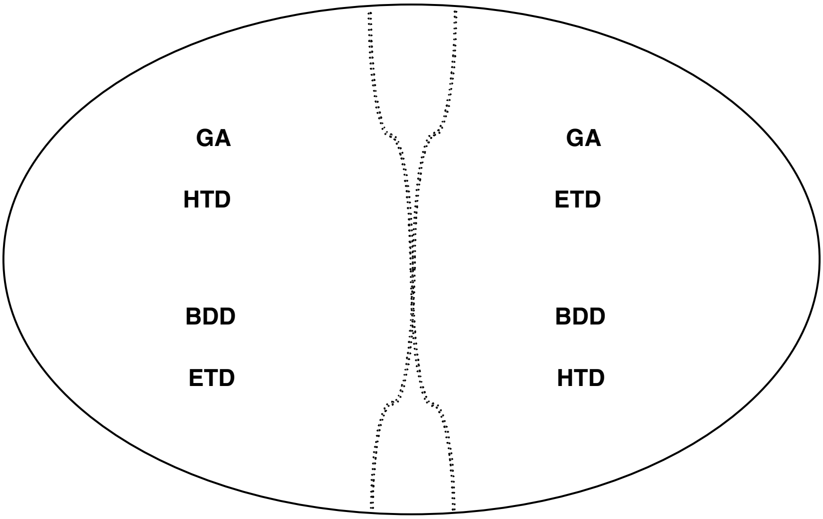


**Figure 3.1.** HTD and ETD errors for a GA-based and a BDD-based ATPG

In conclusion, a GA-based testing technique considers as HTD those faults that are ETD for a BDD-based testing technique and *vice versa*, as shown graphically in Figure 3.1. The two testing approaches are thus complementary and can effectively cooperate. This represents the main motivation for defining a hybrid approach which exploits both random-based ATPG and symbolic-based ATPG to apply the most appropriate technique to HTD and ETD faults.

This work proposes an efficient testing methodology that exploits the previous considerations to obtain the highest coverage with the lowest resource requirements, for each fault class. By mixing BDD-based and random-based functional test generation techniques we realized that they are targeted to different sets of faults; that is, they have disjoined sets of HTD and ETD faults. Particular effort is dedicated to the symbolic aspect of the methodology and it is shown how the symbolic ATPG is integrated in a highly flexible functional testing framework.

The chapter is organized as follows. Section 3.3 is a brief introduction to BDDs. The testing methodology is described in Section 3.4. Section 3.5 is devoted to present the testing framework that has been used for the experimental results reported in Section 3.6. Finally, concluding remarks are summarized in Section 0.

## 3.3 Binary Decision Diagrams

A BDD [3] is a formalism frequently used to implicitly represent digital systems. A BDD is a Directed Acyclic Graph (DAG); the root node of the DAG identifies the function, *f*, represented by the BDD, the internal nodes are labeled with the variables belonging to the true support of *f* (*i.e.*, the set of variables on which factually depends), and the terminal nodes are labeled with the values 0 and 1. As an example, the BDD for the function $f(v_1, v_2, v_3, v_4) = v_1 v_2 v_3 + \overline{v}_1 v_4 + \overline{v}_2 v_4$ is given in Figure 3.2.

A particular kind of BDD is the Reduced Ordered BDD (ROBDD). These do not contain duplicated and redundant nodes, and in addition, they are ordered; that is, all the variables appear in the same order along all paths from the root to the terminal nodes. Given an ordering, the reduced BDD for a function is unique. Hence, a BDD is a canonical representation; that is, two functions *f* and *g* are equivalent (*i.e.*, *f* = *g*) if and only if they have the same BDD.

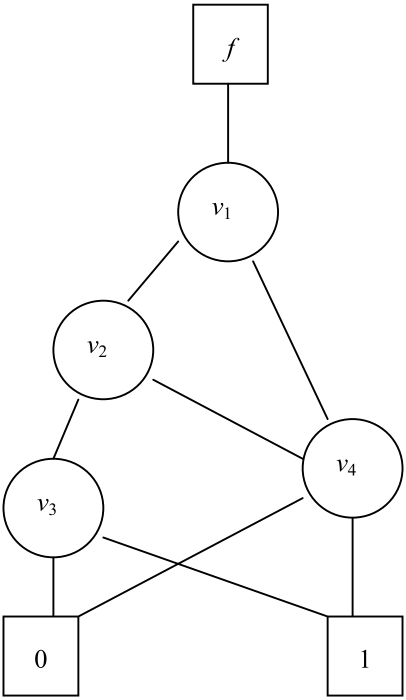


**Figure 3.2.** A BDD for function $f(v_1, v_2, v_3, v_4) = v_1 v_2 v_3 + \overline{v}_1 v_4 + \overline{v}_2 v_4$

## 3.4 Methodology

The main goal of the proposed methodology is the generation of functional test patterns by exploiting a two-way strategy. First, a random-based ATPG session is adopted to address ETD faults. Then, a symbolic ATPG is used to detect the remaining HTD faults. This approach optimizes ATPG time because the exhaustive but time-consuming symbolic session is applied to a limited number of faults that cannot be detected by a faster, random-based, test pattern generator. The methodology is completely implemented into Laerte++, a highly flexible framework for functional testing (see Section 3.5, which reports different ATPG engines).

By using the proposed methodology, functional test patterns can be easily generated for ETD and a large part of HTD faults. In particular, the methodology guarantees:

- fast detection of ETD faults;
- exact identification of redundancies and HTD faults;
- full code coverage for each single process;
- detailed information to detect design errors.

### 3.4.1 The Random-based Approach

Generally, random-based ATPG [14,21] works in the following way:

1. An input sequence is generated in a random way or by exploiting some heuristic. For example, in the case of a GA-based ATPG, heuristics are represented by the fitness function used to guide the test pattern generation.
2. The sequence generated is applied to the fault-free DUT and to all of its faulty instances obtained by injecting, one by one, every fault modeled.
3. The outputs of the fault-free and of the faulty DUTs are compared for every fault modeled. A fault is detected if, for at least one output, the corresponding faulty DUT differs from the fault-free DUT.
4. Steps 1 to 3 are repeated until one of the following conditions is verified:
   - full fault coverage is achieved;
   - the maximum number of input sequences has been generated;
   - the limit for execution time has been reached.

The main problem related to random-based ATPG is represented by lack of exhaustiveness. If a fault remains undetected, we cannot state that the fault is undetectable. In fact, it could be the case that only very few sequences are able to detect that fault, and these have a very low probability to be generated by the ATPG among the infinite space of input sequences. Section 3.5.2 describes how different random-based ATPGs have been defined within the testing framework adopted in this work.

### 3.4.2 The Symbolic Approach

A symbolic-based ATPG session consists of the following three operations:

- Hardware Design Language (HDL) to BDD translation;
- functional test patterns generation for a single process;
- functional test patterns generation for interconnected processes.

### 3.4.3 Hardware Design Language to Binary Decision Diagram Translation

VHDL design entities or SystemC modules are directly converted into BDD-based descriptions. This operation often produces a less complex BDD in relation to the construction of BDDs starting from the corresponding gate-level descriptions [13]. In fact, a smaller number of registers are involved in a high-level description with respect to its implementation, because synchronization registers are not included. Moreover, the device partitioning performed by the designer usually produces unrelated functionalities which depend on few input/output variables. This is a good criterion for building small BDDs, and the same operation is hard to perform on a flat gate-level description [4]. However, whenever the size of the BDD increases considerably (*i.e.*, in the case of circuits including large multipliers) we adopt approximate techniques.

### 3.4.4 Functional Vector Generation for a Single Process

Functional vectors identification is based on test pattern generation techniques [11,12]. A fault model based on signal/variables stuck-at is used to identify functional vectors and possibly code redundancies. The fault model adopted has been proved [2] to model Register-Transfer-Level (RTL) and gate-level stuck-at faults accurately. A fault is certified as *behaviorally redundant* if no vector exists that distinguishes the fault-free BDD-based representation from the faulty BDD. Moreover, the behavioral specification is decomposed into several BDD-based descriptions with reduced size. Hence, the test generation solution is to apply the test generation algorithm multiple times to sub-problems of affordable size, whose solution may be found efficiently. The test generation algorithm works on a BDD-based description searching a test vector in a restricted domain, whose size is incrementally expanded until the entire domain space is explored. This incremental analysis is very efficient because it allows an early removal from the fault list of all the simple faults, *i.e.*, all faults that are tested by several test vectors, leaving the analysis of all HTD faults at the end. This is a scalable approach, allowing the identification of test vectors in a very large number of cases with a reduced time complexity and memory size. However, this domain decomposition implies the possible identification of aborted faults that could be, in principle, either redundant or testable. We can therefore classify as *behaviorally redundant* faults only if the entire input domain has been searched, whereas the other faults are classified as HTD. Whenever all the strategies adopted fail to provide a manageable BDD description of a process, faults which remain untested are classified as HTD. Our approach provides the exact code location of the fault, thus allowing the designer to inspect that portion of code to identify a potential design error.

### 3.4.5 Functional Vector Generation for Interconnected Processes

Each process can be correctly designed and simply verified in isolation, but when connected to other processes can produce design errors due to incorrect connections or overspecification. This analysis related functional test generation for a single process to a controllability analysis among a network of processes [10].

To take into account the interaction among processes, we compute a new type of controllability set for each process. The concept of a *controllability-don't-care set* has been defined in logic synthesis [8] as the set of primary input combinations that never occur. Two sets can be identified: *external* controllability-don't-care set ($CDC^{ext}$) and *internal* controllability-don't-care set (CDC). The first one is the set containing the input patterns never produced by the environment at the network's inputs, while the second one refers to internal nodes. To compute the internal CDC set, the network is traversed by considering different cuts moving from the inputs to the outputs. A CDC set is defined for each cut, corresponding to the bit vectors never applied to the nets traversed by the cut. Similar considerations can be made for the observability-don't-care (ODC) set. CDC and ODC sets are used for logic optimization and for the synthesis of testable circuits at gate level [8]. The concept of a CDC set can be exploited in the analysis of interacting processes, for functional testing, because it provides the patterns that are applicable to a process.

## 3.5 The Testing Framework

The testing methodology proposed in this work is completely implemented into Laerte++, which is a functional testing framework for SystemC/VHDL designs. It tries to address testing issues within the whole design flow of a digital system, from the system-level description to its structural representation, by using the same fault model and the same test generation technique.

Laerte++ architecture is based on standard template library (STL) data containers [20] and native SystemC data types. In this way, the majority of software structures, necessary to implement this testing framework, are directly built by exploiting SystemC language characteristics.

The main features of Laerte++ are:

- ***Fault model.*** C++ abstract classes are provided in order to model a wide set of fault models and to evaluate the coverage of the test patterns applied. Each one of these classes implements a different saboteur function that is able to modify the behavior of the DUT according to the related fault model. The DUT is modified by automatically adding instances of the desired saboteur function to obtain a faulty description of the DUT.
- ***Design introspection capabilities.*** Internal signals and memory elements can be directly observed and controlled from the testbench. This information is used to guide the test pattern generation efficiently.

- ***ATPG engine.*** A set of symbolic and random-based test pattern generators, which can be customized for the design under test, are defined. Every ATPG engine generates a test pattern and measures the fault coverage according to the selected fault model.
- ***Hierarchical test support.*** The introspection capabilities are exploited for defining soft-wrappers around internal design modules and for applying a hierarchical test procedure.

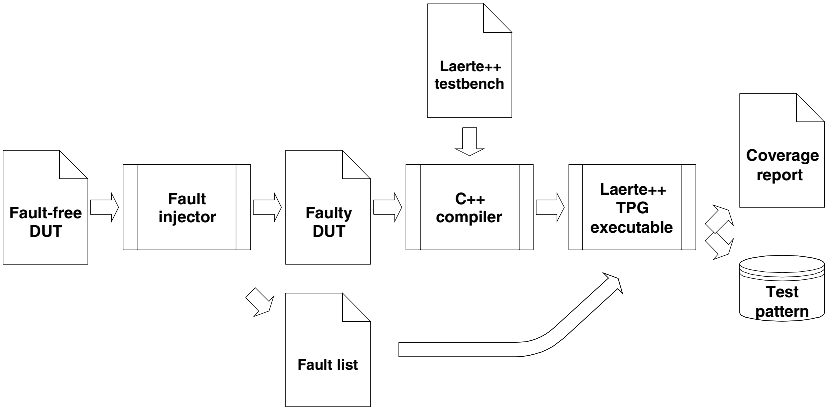


**Figure 3.3.** Laerte++ setup flow

Figure 3.3 shows the procedure to set up a test session based on the Laerte++ framework. The first stage is completed by the fault injector, which generates the faulty DUT description. This description is then connected to the defined Laerte++ testbench and jointly compiled to obtain a single executable program for generating test patterns for the DUT.

The effort required for defining an *ex novo* testbench should be considered as an important parameter to evaluate the effectiveness of a testing environment. The code reported in Figure 3.4 shows how to define in Laerte++ a complete testbench for a DUT. The testbench definition for the example considered requires very few additional C++ code lines, which are so simple to be almost self-explanation. The DUT is instantiated, as usual in SystemC, then names of its input–output ports are passed to Laerte++ with some other information related to the clock and reset signals; finally, a test pattern generation session is activated. The compilation of the main code produces a single executable program.

```
#include <laerte.h>
#include “dut.h”

int sc_main(int ac, char* av[]) {

   sc_signal<int>          injfault;
   Lrt_signal<sc_logic>    reset(1);
   Lrt_signal<sc_logic>    dutclk(1);
   Lrt_signal<sc_logic>    start(1);
   Lrt_signal<sc_lv<16> > data_in(16);
   Lrt_signal<bool>        wr(1);
   Lrt_signal<sc_lv<16> > accout(16);
   Lrt_signal<sc_lv<10> > addr(10);
   cdut = new cpu_rtl_autl(“module”); //DUT instantiation
   cdut->start(start);
   cdut->clear(clear);
   cdut->clk(dutclk);
   cdut->data_in(data_in);
   cdut->wr(wr);
   cdut->accout(accout);
   cdut->addr(addr);
   cdut->fault_port(injfault);
   Laerte laerte(ac, av); //Command line parameter parsing
   laerte.record_fault(&injfault);//Fault signal registra-
tion
   laerte.record_reset(&reset); // Reset & Clock registra-
tion
   laerte.record_clock(&dutclk);
   laerte.record_pi(&start);   // In/Out signals registra-
tion
   laerte.record_pi(&data_in);
   laerte.record_pi(&wr);
   laerte.record_cpo(&accout);
   laerte.record_cpo(&addr);

   laerte.init();           // Laerte++ initialization
   laerte.run();            // Simulation/Test execution
   laerte.print_report(&cout);      // Print results
}
```

**Figure 3.4.** Laerte++ testbench

### 3.5.1 Fault Model Definition

The aim of the proposed testing methodology is to detect design errors through functional testing. Thus, a fault model is necessary to simulate the effects of design errors on the DUT and to estimate its testability.

Laerte++ defines the pure abstract `FaultBase` class to represent a generic fault model (see Figure 3.5). The function `inject` acts as a saboteur that supplies the fault free or the faulty value of the target object according to the value of a control line. Instances of this function are automatically injected into the description of the DUT to generate the faulty description (see section The Fault Injection Strategy). Different fault models can be implemented by redefining the `inject` function. The BitFault class extends the FaultBase class to implement the *bit coverage* fault model which is adopted in this work (see section The Bit Coverage Fault Model).

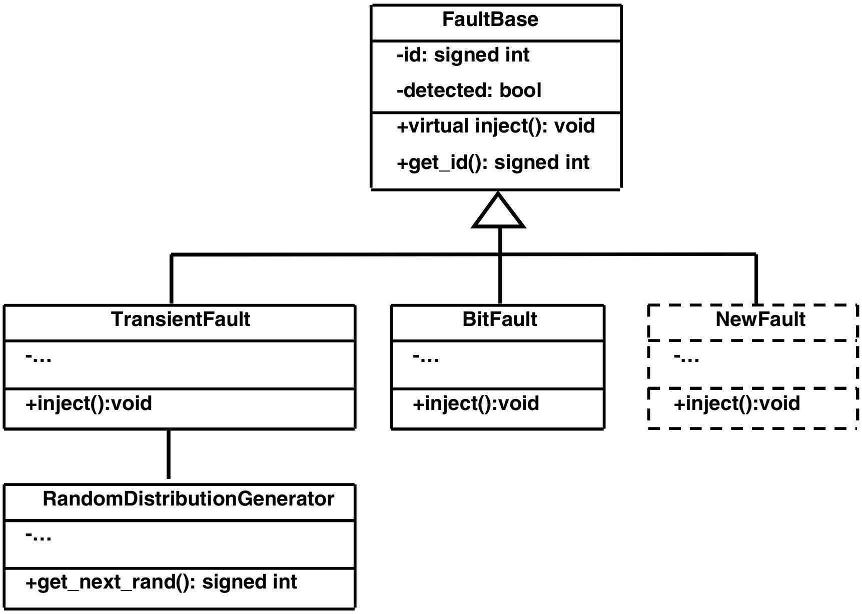


**Figure 3.5.** Fault hierarchy

New fault models can be simply defined by deriving them from the FaultBase class: it is only necessary to implement the virtual method inject (see Figure 3.5). For instance, let us imagine the defining of a complex transient fault model. It is sufficient to identify a C++ class for random generation and to select a random distribution (*e.g.*, normal, Poisson, *etc.*) to define in which time frame the fault is active.

```
class TransientFault : public FaultBase {
  public:
      void inject() {
          if (poisgen->next() > threshold) {
              apply_fault();}}
   private:
      void apply_fault();
      PoissonGen* poisgen;
      void*       faulttarget;
}
```

**Figure 3.6.** TransientFault class definition

Figure 3.6 shows the definition of a transient fault model, which has a transient behavior modeled by a Poisson distribution.

### *The Bit Coverage Fault Model*

The fault model adopted in this work is the *bit coverage*. During fault simulation or test pattern generation, at most a one bit coverage fault can be activated according to the following failure specification:

- ***Bit failures.*** Each occurrence of variables, constants, signals or ports is considered as a vector of bits. Each bit can be stuck-at-0 or stuck-at-1 independently from its real value.
- ***Condition failures.*** Each condition can be stuck-at-true or stuck-at-false independently from its real value, thus removing some execution paths in the faulty DUT representation.

The original definition of bit coverage [11] directly covers all language characteristics of VHDL and SystemC 1.2, which model RTL descriptions. On the contrary, it has been extended to cover the new SystemC 2.0 language features concerning events and channels as follows [16]:

- ***Event failures.*** Events are faulted in two different ways: by changing the eventual parameter of the `notify` method and by avoiding the event notification. Parameter modification is already modeled by the bit coverage fault model, since the parameter is an `sc_time` variable or constant. On the other hand, notification avoidance is obtained inserting an extra conditional statement.
- ***Channels failures.*** Channels are similar to ports and signals from the fault model point of view. They can be faulted by modifying data managed by channels methods. Such data are faulted by following the bit failure strategy of the bit coverage fault model.

Bit coverage can be easily related to the other metrics, developed in the software engineering field [19] and commonly used in functional testing. In particular, it unifies into a single metric the well-known statements, branches and conditions coverage. An important part of all paths is also covered and all blocks of a description are activated several times.

- ***Statement coverage.*** Any statement manipulates at least one variable or signal. The bit failures are injected into all variables and signals on the left-hand and right-hand side of each assignment. Thus, at least one test vector is generated for all statements. To reduce the proposed fault model to statement coverage it is thus sufficient to inject only one bit failure into one of the variables (signals) composing a statement. In conclusion, the bit coverage metric induces an ATPG to produce a larger number of test patterns with respect to statement coverage and it guarantees to cover all statements.
- ***Branch coverage.*** The branch coverage metric implies the identification of patterns which verify the execution of both the true and false (if present) paths of each branch. Modeling of our condition failures implies the identification of patterns which differentiate the true behavior of a branch from the false behavior, and *vice-versa*. This differentiation is performed by making stuck at true (false) the branch condition and by finding patterns executing the false (true)

branch, thus executing both paths. In conclusion, the proposed bit coverage metric includes the branch-coverage metric.

- ***Condition coverage.*** The proposed fault model includes condition failures which make stuck at true or stuck at false any condition disregarding the stuck-at values of its components.
- ***Path coverage.*** The verification of all paths of a SystemC method can be a very complex task owing to the possible exponential grow of the number of paths. The proposed fault model selects a finite subset of all paths to be covered. The subset of covered paths is composed of all paths that are examined to activate and propagate the injected faults from the inputs to the outputs of the SystemC design module within a given time limit.
- ***Block coverage.*** In [7] statement coverage has been extended by partitioning the code in blocks and by activating these blocks a fixed number of times. This block coverage criterion is included in the proposed fault model in case the number of bit faults included in a block is larger than the number of times the block is activated. In fact, a test pattern is generated for each bit fault; thus, the block including the fault is activated when the fault is detected.

### The Fault Injection Strategy

Faults are automatically injected into the design by using an automatic injection technique which is independent from the HDL adopted to describe the DUT [15,17]. This allows us to describe the DUT in VHDL as well as in SystemC.

Fault injection is performed by inserting saboteurs (`inject` function) into a language-independent intermediate representation of the design. The first version of fault injector exploited the In-memory Intermediate Representation (IIR) of the Savant environment [9], while the more recent one is based on the commercial Arexsys Internal Format (AIF) of Vista [18]. The migration from IIR to AIF allows one to manage the new features of SystemC 2.0 not supported by Savant. In both cases, front-end and back-end tools for VHDL and SystemC, in conjunction with an accurate API library, allow us simply to manipulate IIR or AIF descriptions to inject faults in VHDL/SystemC code. After fault injection, the intermediate representation of the DUT is converted in SystemC to be linked with one of the ATPG engines of Laerte++.

```
FUNCTION inject_fault_bit
   (object: BIT; fault_code: INTEGER; start_s0: INTEGER;
    end_s0: INTEGER; start_s1: INTEGER; end_s1: INTEGER)
   RETURN BIT IS
   VARIABLE res: BIT;
BEGIN
   IF (fault_code = start_s0) THEN
     res := '0';
   ELSIF(fault_code = start_s1) THEN
     res := '1';
   ELSE
     res := object;
   END IF;
   RETURN res;
END;
```

**Figure 3.7.** Saboteur VHDL function for bit operands

Every occurrence of signal, variable, constant and condition within statements of the high-level description of the DUT is replaced by an appropriate bit coverage saboteur. They are functions which can supply the correct or faulty value of the corresponding object depending on the value of a control signal. For the bit coverage fault model, a saboteur for every language type, *i.e.*, bit, integer, standard logic, Boolean, *etc.* has been defined. Faults are enumerated and an integer-type port, named `fault,` is added to the design. The `fault` port drives all control signals of saboteurs. Figure 3.7, Figure 3.9 and Figure 3.8 respectively show the saboteur function for VHDL `bit`, `bit_vector` and `integer` operands. Saboteurs for other data types are defined in a similar way referring to the bit case, and analogous functions are defined for SystemC.

```
FUNCTION inject_fault_integer
   (object: INTEGER; fault_code: INTEGER; start_s0:
INTEGER;
    end_s0: INTEGER; start_s1: INTEGER; end_s1: INTEGER)
    RETURN INTEGER IS
    VARIABLE length : INTEGER;
    VARIABLE res: INTEGER;
BEGIN
   length := end_s0 - start_s0 + 1;
   res := to_int(inject_fault_bit_vector(
                 to_bit_vector(object, length),
                 fault_code, start_s0, end_s0,
                 start_s1, end_s1));
   RETURN res;
END;
```

**Figure 3.8.** Saboteur VHDL function for integer operands

```
FUNCTION inject_fault_bit_vector
 (object: BIT_VECTOR; fault_code: INTEGER; start_s0:
INTEGER;
  end_s0: INTEGER; start_s1: INTEGER; end_s1: INTEGER)
   RETURN BIT_VECTOR IS
   VARIABLE left       : INTEGER;
   VARIABLE right      : INTEGER;
   VARIABLE index      : INTEGER;
   VARIABLE index_s0   : INTEGER;
   VARIABLE index_s1   : INTEGER;
   VARIABLE length     : INTEGER;
   VARIABLE res_downto: BIT_VECTOR(end_s0-start_s0 DOWNTO
0);
   VARIABLE res_to     : BIT_VECTOR(0 TO end_s0 -
start_s0);
BEGIN
   left := object'left;
   right := object'right;
   length:= end_s0 - start_s0;
   IF (left > right) THEN        -- vector range is downto
```

```
     res_downto := object;
     FOR index IN length DOWNTO 0 LOOP
       index_s0 := index + start_s0;
       index_s1 := index + start_s1;
       res_downto(index) := inject_fault_bit(object(index),
                            fault_code, index_s0, index_s0,
                            index_s1, index_s1);
     END LOOP;
     RETURN res_downto;
   ELSE                             -- vector range is to
     res_to := object;
     FOR index IN 0 TO length LOOP
       index_s0 := index + start_s0;
       index_s1 := index + start_s1;
       res_to(index) := inject_fault_bit(object(index),
                        fault_code, index_s0, index_s0,
                        index_s1, index_s1);
     END LOOP;
     RETURN res_to;
   END IF;
END;
```

**Figure 3.9.** Saboteur VHDL function for bit vector operands

Considering the signature of the saboteur functions, the parameter `object` is the target of the fault, while `fault_code` is the value of the `fault` port. Parameters `start_s0-1` and `end_s0-1` show the range for `fault_code` to activate the stuck-at 0-1 on the target object. The fault injection process generates a unique faulty description of the design that includes all bit coverage faults. Figure 3.10 shows an example of fault-free and faulty VHDL descriptions by using bit coverage saboteurs. It illustrates how the faults are inserted recursively in complex statements as an `if-then-else` statement. For example, to activate the fault stuck-at 0 on the third bit of the integer signal `rmax` the signal `fault` must be set to 1456, since the range for faults stuck-at 0 on `rmax` is from 1454 to 1461. On the other hand, to activate the fault stuck-at true on the `if-then-else` condition the signal `fault` must be set to 1473.

```
IF (data_in > rmax) THEN
   ack <= '1';

IF (inject_fault_bool(
   inject_fault_integer(data_in,fault,1438,1445,1446, 1453)
>
   inject_fault_integer(rmax, fault, 1454, 1461, 1462,
1469),
   fault, 1470, 1470, 1471, 1471)) THEN
     ack <= inject_fault_bit('1',fault,1472,1472,
1473,1473);
```

**Figure 3.10.** Fault-free and generated faulty VHDL code

### 3.5.2 Automatic Test Pattern Generation Engines

The core of the ATPG engine is the pure abstract class `Sequence`. It is the base class for all kinds of derived sequence classes; see Figure 3.11.

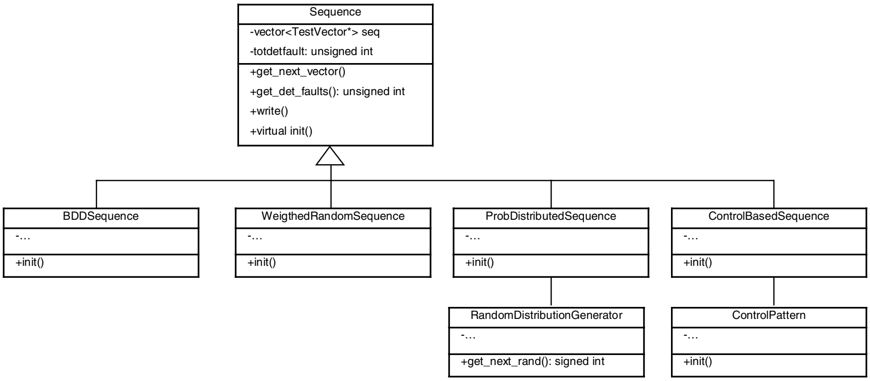


**Figure 3.11.** Sequence hierarchy

In the current Laerte++ implementation, five subclasses have been derived. Four of them implement random-based ATPG algorithms and one implements a symbolic BDD-based approach:

- ***Weighted random sequences.*** The virtual method `init` of the `Sequence` class is implemented to generate test sequences with a predefined percentage $\alpha$ of bits set to 1 and $(1 - \alpha)$ bits set to 0.
- ***Probabilistic distributed sequences.*** Test sequences with the most appropriate distribution can be generated by including into the `ProbDistributedSequence` class a random number generator. Such a random number library allows us to define random permutations. This feature can be efficiently exploited for generating a DUT highly tailored test sequences. For example, a CPU can be tested by generating opcodes with a required distribution (*e.g.*, 40% of memory access, 30% of logic operations, 10% of arithmetic instructions and 20% of stack access). This allows us to test the DUT with test sequences approximating such software applications executed on it.
- ***Control-based sequences.*** This `Sequence` sub-class allows us to explicitly define the value of the bits related to the control PIs of the DUT, in the case where a partition between control PIs and data PIs can be done. Whenever control PIs are set to some constrained values, and data PIs are unspecified, the `init` method completes the unspecified bits of a test sequence by generating random values for the data PIs only, thus preserving control values. The `ControlBasedSequence` class allows us to test the same functionality with alternative input data in order to analyze specific design behaviors deeply.
- ***GA-based sequences.*** Most of the high-level ATPG engines based on GAs [5,6] can be reproduced by exploiting combinations of the Laerte++ features. In fact, fitness functions, which require one to sample the memory element's ac-

tivity, can exploit the `RegWatcher`. Moreover, the methods `control` and `observe` of the `Lrt_signal<>` class can be used to measure the target fault propagation effects and can be exploited for defining complex fitness functions. For example, the fitness functions presented in [5,6]:

- $f(s) = \sum_{i}^{\text{BasicBlock}} \text{ExecutionCount}(s, \beta_i)$
- $f(s) = \sum_{i}^{\text{BasicBlock}} \text{ExecutionCount}(s, \beta_i) c_{it}$
- $f(s) = \sum_{f \in \Omega_i} k_1 \text{Det}(f) + k_2 \text{Ons}(f) + k_3 \text{Exc}(f)$

where $\beta_i$ is the activated basic block, $c_{it}$ is the correlation factor for the basic block $i$ and the target fault $t$, where Det($f$), Obs($f$) and Exc($f$) are respectively the number of detected faults, internally observed fault effects and propagated fault effects. $k_1$, $k_2$ and $k_3$ are the assigned weights to such factors. They can be reproduced by exploiting `Lrt_signal<>` for measuring fault effect propagation. The fault effects on the memory elements are observed by the `RegWatcher` objects.

- ***BDD-based sequences.*** This class implements a BDD-based ATPG [12] which adopts the symbolic approach described in section 3.4.2.

### *New Test Sequence Definition*

The C++ class inheritance paradigm allows the easy definition of new sequence classes by single or multiple class derivation. For instance, a control-based sequence with a Gaussian probabilistic distribution for the input data values can be obtained by deriving it from the `ControlBasedSequence` class, but adding a reference to a Gaussian random generator. Figure 3.12 shows the source code sketching it.

```
class ControledGaussianSequence : public ControlBasedSe-
quence
{
      public:
      void init() {
          Control-BasedSequence::init();
          InitDataBit();
      }
      void InitDataBit();
          ...
      private:
      GaussianGen* gausgen;
}
```

**Figure 3.12.** A new sequence class definition

## 3.6 Experimental Results

The effectiveness of the proposed methodology has been evaluated on some benchmarks, whose characteristics are summarized in Table 3.1. For each benchmark three different data sizes (8, 16, and 32) have been considered as shown in Column 2. Columns 3–5 report respectively the number of bits for PIs, primary outputs and internal signals and variables. Then Column 5 and Column 6 show the number of gates and memory elements for every design. The number of bit coverage injected faults is indicated in the last column.

**Table 3.1**. Characteristics of the benchmarks analyzed

| Name | BUS-size | In.bits | Out.bits | Int. bits | Gates | FFs | Faults |
|---|---|---|---|---|---|---|---|
| diffeq | 8 | 42 | 24 | 104 | 2012 | 100 | 5188 |
|  | 16 | 82 | 48 | 128 | 3438 | 180 | 10008 |
|  | 32 | 162 | 96 | 256 | 9520 | 356 | 24196 |
| ellipf | 8 | 66 | 64 | 232 | 2239 | 141 | 5714 |
|  | 16 | 130 | 128 | 464 | 4684 | 277 | 11260 |
|  | 32 | 258 | 256 | 928 | 9244 | 549 | 20764 |
| fir | 8 | 194 | 8 | 40 | 2233 | 118 | 6482 |
|  | 16 | 384 | 16 | 80 | 4570 | 262 | 14704 |
|  | 32 | 768 | 32 | 160 | 8827 | 204 | 23672 |
| gcd | 8 | 18 | 8 | 24 | 636 | 35 | 1588 |
|  | 16 | 34 | 16 | 48 | 1143 | 67 | 3338 |
|  | 32 | 66 | 32 | 96 | 2153 | 131 | 6386 |

The genetic-based engine of Laerte++ has been used to generate test patterns and achieve an initial estimation of the fault coverage. Most of the injected faults are detected by the GA-based ATPG; however, a certain number of faults remain undetected. We consider these faults HTD for random-based ATPG (RAND-HTD). Then, a further ATPG session has been launched by exploiting the BDD-based engine to detect only these RAND-HTD faults. Actually, the BDD-based engine sensibly increases fault coverage, spending a low amount of time for all benchmarks. These results empirically show that the most part of RAND-HTD are ETD for a symbolic approach. Experimental results are summarized in Table 3.2. Column 2 and Column 3 show the fault coverage and the execution time related to the GA-based ATPG. Column 4 shows the fault coverage achieved after the BDD-engine was applied to detect remaining *RAND-HTD* faults, while Column 5 reports the execution time required by the BDD-engine to generate test patterns for *RAND-HTD* faults. The total ATPG time (RANDOM+BDD) appears in Column 6.

**Table 3.2.** Random-based ATPG versus BDD-based ATPG

| Name | %FC | Time (s) | %FC | Time (s) | Total time (s) |
|---|---|---|---|---|---|
| diffeq_8 | 92.9 | 1200.2 | 98.1 | 6.8 | 1207.0 |
| diffeq_16 | 95.2 | 640.7 | 98.9 | 19.9 | 660.6 |
| diffeq_32 | 96.2 | 5630.4 | 99.7 | 114.3 | 5744.7 |
| ellipf_8 | 93.9 | 245.9 | 98.4 | 2.1 | 248.0 |
| ellipf_16 | 94.6 | 476.2 | 99.4 | 34.5 | 510.7 |
| ellipf_32 | 94.4 | 1438.9 | 99.2 | 78.6 | 1517.5 |
| fir_8 | 80.8 | 793.4 | 91.8 | 30.8 | 824.2 |
| fir_16 | 66.7 | 8637.0 | 89.2 | 246.9 | 8883.9 |
| fir_32 | 96.3 | 5616.7 | 98.7 | 116.8 | 5732.8 |
| gcd_8 | 92.8 | 289.9 | 98.9 | 1.7 | 291.6 |
| gcd_16 | 64.1 | 890.0 | 96.0 | 211.3 | 1101.3 |
| gcd_32 | 67.5 | 4020.1 | 96.2 | 675.4 | 4695.5 |

FC: Fault Coverage.

## 3.7 Concluding Remarks

This chapter presents a functional testing environment that exploits a BDD-based engine to increase the fault coverage achieved by random-based ATPGs. Considerations about the different meaning of HTD and ETD faults with respect to the ATPG approach adopted have been explained. In particular, we have shown how ETD faults for a random-based ATPG are HTD for a BDD-based ATPG and *vice versa*. The experimental results strengthen this conjecture, and they show how a hybrid approach where a BDD ATPG session is applied to faults not detected by a random-based ATPG allows us to sensibly increase fault coverage. This hybrid approach has been implemented in the Laerte++ framework, where object-oriented principles and native SystemC characteristics have been used to simplify the set-up and run of ATPG sessions. Basic tasks of any functional ATPG have been implemented by extending standard SystemC classes. This includes testbench set-up, fault models, run-time coverage measurements and ATPG engines. This deep integration allows the setting up and running of an *ex novo* ATPG session by adding very few C++ code lines to any DUT description.

## Acknowledgments

We would like to thank Fabrizio Ferrandi, Donatella Sciuto and Alessandro Fin for their contributions on previous versions of this work.

## References

[1] Breuer MA, Abramovici M, Friedman AD (1990) Digital systems testing and testable design. IEEE Press

[2] Buonanno G, Ferrandi L, Ferrandi F, Fummi F, Sciuto D (1997) How an evolving fault model improves the behavioural test generation. In: Proc. of ACM Great Lake Symposium on VLSI, 124-129

[3] Bryant R (1986) Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers, 35(8): 79-85

[4] Cho H, Hachtel GD, Macii E, Plessier B, Somenzi F (1993) Algorithms for approximate FSM traversal based on space state decomposition. In: Proc. of ACM/IEEE Design Automation Conference, 25-30

[5] Corno F, Cumani G, Sonza Reorda M, Squillero G (2000) An RT-level fault model with high gate level correlation. In: Proc. of IEEE International High Level Design Validation and Test Workshop, 3-8

[6] Corno F, Cumani G, Sonza Reorda M, Squillero G (2001) Arpia: A high-level evolutionary test signal generator. In: Proc. of IEEE Workshop on Evolutionary Image Analysis, Signal, Processing and Telecommunications, 298-306

[7] Corno F, Prinetto P, Sonza Reorda M (1997) Testability analysis and ATPG on behavioural RT-level VHDL. In: Proc. of IEEE Int. Test Conference, 753-759

[8] Damiani M, De Micheli G (1993) Don’t care set specification in combinational and synchronous logic circuits. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 12(3): 365- 388

[9] Dept. of ECECS, University of Cincinnati (1999) Savant programmer's manual. Technical report

[10] Ferrandi F, Fummi F, Pravadelli G, Sciuto D (2003) Identification of design errors through functional testing. IEEE Transactions on Reliability, 52(4): 400-412

[11] Ferrandi F, Fummi F, Sciuto D (1998) Implicit test generation for behavioural VHDL models. In: Proc. of IEEE Int. Test Conference, 436-441

[12] Ferrandi F, Fummi F, Sciuto D (2002) Test generation and testability alternatives exploration of critical algorithms for embedded applications. IEEE Transactions on Computers, 51(2): 200-215

[13] Ferrandi F, Fummi F, Macii E, Poncino M, Sciuto D (1996) BDD-based testability estimation of VHDL designs. In: Proc. of IEEE European VHDL Conference, 444-449

[14] Fin A, Fummi F (2003) LAERTE++: An object oriented high-level TPG for SystemC designs. In: Proc. of ECSI Forum on Design Languages

[15] Fin A, Fummi F, Pravadelli G (2001) Amleto: A multi-language environment for functional test generation. In: Proc. of IEEE Int. Test Conference, 821-829

[16] Fin A, Fummi F, Pravadelli G (2003) SystemC: methodologies and application. In: SystemC as a Complete Design and Validation Environment, Kluwer Academic Publishers, 127-156

[17] Fummi F, Marconcini C, Pravadelli G (2003) Redundant functional faults reduction by saboteur synthesis. In: Proc. of IEEE Int. High Level Design Validation and Test Workshop, 108-113

[18] Moussa I, Grellier T, Nguyen G (2003) Exploring SW performance using SOC transaction-level modelling. In: Proc. of IEEE Design Automation and Test in Europe, 120-125

[19] Myers GJ (1979) The Art of Software Testing. Wiley-Interscience, New York

[20] Silicon Graphics. Standard Template Library Documentation. http://www.sgi.com/tech/stl/.

[21] Yu X, Fin A, Fummi F, Rudnick EM (2002) A genetic testing framework for digital integrated circuits. In: Proc. of IEEE Int. Conference on Tool with Artificial Intelligence, 521-526

# 4 Test Generation: A Heuristic Approach

O. Goloubeva, M. Sonza Reorda, M. Violante

Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy

## 4.1 Abstract

The adoption of the System-on-Chip design paradigm creates new challenges for designers and test engineers. In this chapter a high-level test generation approach is presented, which is able to produce high-quality vectors that can be fruitfully exploited for test and validation purposes of both the hardware and software components of System-on-Chip designs. Experimental results are reported showing that our high-level test generation algorithm produces high-quality vectors in terms of stuck-at fault coverage for hardware components and code mutants for software components. The vectors produced can also be exploited for validation purposes, as the results gathered while validating a processor core suggest.

## 4.2 Introduction

In recent years, new techniques have been developed to integrate an entire system on a single chip, resulting in a new design paradigm known as System-on-Chip (SOC). SOC products represent a real challenge not only from the manufacturing point of view, but also when design issues are concerned.

To cope with the challenges faced by SOC designers, tools and techniques dealing with design at high levels of abstraction are becoming an industrial reality. Thanks to the availability of design tools supporting the system level of abstraction, most of the implementation details can be neglected; the designers can thus focus their effort on the definition of the system behavior that best fits with the user needs and analyze the cost/benefit trade-off of any given solution.

In particular, behavioral-level synthesis tools and the more recently introduced co-design environments are starting to play an important role in the initial phases of the design process. The major benefit stemming from these design environments is the possibility of quickly evaluating the costs and benefits of different architecture alternatives, including both hardware and software components, starting from the algorithm on SOC should implement.

While the design practice is quickly moving toward higher levels of abstraction, test issues are still mainly considered only when a detailed description of the design is available. For the hardware modules composing SOCs, the test is typically addressed at the gate level for test-sequence generation and at the Register-Transfer (RT) level for design-for-testability structure insertion. As far as software modules are concerned, the exploitation of *formal verification* techniques has been advocated by some authors [9] as a viable solution for assessing their correctness, but its widespread adoption is limited by the overhead, both in terms of expertise and resource, it implies. Several approaches have been proposed to overcome these limitations. *Symbolic evaluations* [7,17] consist in first assigning symbolic values to variables; the paths composing the program control flow graph are then traversed, and a list of symbolic representations of each condition predicate along the paths is recorded. Through the analysis of the constraints obtained it is then possible to determine if paths are executable as well as the conditions for their activation. Simpler approaches are those based on *path testing* [14] and *test coverage metrics* [1], where the number of paths, branches and statements executed are counted, and those based on *functional testing* [1,15], where programs are considered as black boxes and the outputs they produce in response to a set of input stimuli are checked for conformity with the expected behavior. Finally, an approach inspired by those adopted in the hardware test community is *mutation testing* [8,19]. During mutation testing, faults are introduced into a program by creating many versions of the program, each of which contains one fault. A faulty program is thus a *mutant* of the original one. The programs obtained are then executed under a given set of input stimuli to observe which faults produce wrong results. Usually, input stimuli used during the mutation-testing process are either provided by designers, or they are randomly generated within a Monte Carlo simulation procedure. As a result, the input stimuli do not guarantee complete coverage of all the mutants possibly affecting the software tested.

Today, the aforementioned scenario is rapidly changing. The introduction of system-level design tools has unified the process of specification of the software and hardware modules of SOCs. As an example, the users of co-design tools start the design process by just describing the system behavior, exploiting a system-level specification language such as SystemC [23], while neglecting which module will be implemented in hardware and which one in software. In this scenario, provided that a test generation process is available to deal with behavioral specifications, the automatic generation of a single set of input stimuli suitable for testing both hardware and software becomes possible.

The availability of an effective behavioral-level test generation process mandates the definition of suitable fault models and test generation algorithms supporting them.

As far as the fault model is considered, the following characteristics should be met. On the one hand, the fault model must be applicable to behavioral specifications *e.g.* composed of variable assignments, arithmetic/logical operations among variables and control flow instructions. On the other hand, the fault model should be representative of faults that can affect either the hardware or the software components of SOCs. Several high-level fault models can be found in the literature, which are inspired by those known in the software testing [1] domain, and that extend them to cope with hardware descriptions. The state of the art of high-level fault models is described in [12], where *bit failures* and *conditions failures* are used to model faults affecting the memory elements and the control logic of hardware components, while only their behavioral specifications are known.

As far as the test generation algorithm is considered, several approaches have been proposed in the past that are able to maximize the coverage figure of the high-level fault model adopted. Most of them are able to generate test patterns of good quality, sometimes comparable or even better than those of gate-level Automatic Test Pattern Generation (ATPG) tools. However, lacking general applicability, these approaches are still not accepted by the industry. The different approaches are based on different assumptions and on a wide spectrum of distinct algorithmic techniques. Some are based on extracting from a behavioral description the corresponding control machine [18] or the symbolic representation based on binary decision diagrams [11], while others also synthesize a structural description of the data path [10]. Some approaches rely on a direct examination of the Hardware Design Language (HDL) description [3], or exploit the knowledge of the gate-level implementation [21]. Some others combine static analysis with simulation [6,22].

The common denominator of the approaches proposed already is that they only aim at generating input stimuli targeting faults affecting hardware components. They are indeed intended to be exploited in design flows based on behavioral synthesis, where the system is a purely hardware one. Furthermore, the goodness of the produced input stimuli is evaluated as the gate-level stuck-at fault coverage they attain.

In this chapter we propose a high-level test generation approach which is able to provide input stimuli (called *test vectors* in the following) that can be effectively used for testing both hardware [16] and software modules.

In our work, we target purely behavioral system descriptions, where the behavior of each module composing the SOC is coded resorting to the SystemC specification language. The approach we propose exploits a high-level fault model for driving a test generation procedure, which is performed by a simple test generation algorithm based on a heuristic search. Effective test vectors can be generated without any knowledge about the system tested except its behavior. Moreover, it can be used to *refine* input vectors already available (*e.g.*, provided by designers). During the design process the designers already produce many input vectors, which are used in the simulations required to assess the correctness of the implementation. These hand-produced vectors are very powerful, since they are intended for highlighting bugs the designers may introduce during the development process; they can thus also be fruitfully exploited in the test phase. Obviously, the input vectors the designers produce are not exhaustive, since subtle bugs may exist that cannot be easily identified. Moreover, hardware faults may correspond to the modification of system behavior that is too complex to be dealt with by hand by designers. As a result, an automatic procedure for test vectors generation is still required (as well as a fault model supporting it), but it can greatly benefit from exploiting vectors that already exist.

In this chapter we first describe the assumptions at the base of our work, as well as the test generation algorithm we developed. We then propose two possible applications of vectors generated at the high level while neglecting the implementation of the behavioral specification we are dealing with.

The first application is the test of hardware/software systems, where high-level generated vectors are used to discover manufacturing errors in the hardware modules and bugs in the software modules. To investigate the effectiveness of the high-level generated vectors better, we also report coverage figures obtained by a classical ATPG tool working at the gate level.

The second application we present is the validation of application-specific processors. These are processors that have been customized for running software implementations of the behavioral specification for whom test vectors have been computed.

## 4.3 Assumptions

The system we consider is modeled as a network of modules that are described in a purely behavioral fashion. Variable assignments, arithmetic and logical operations among variables, and control flow instructions are used to describe how the modules react to a set of input stimuli to produce the corresponding output values.

The communication among modules is assumed to be synchronous. Moreover, given a set of input stimuli arranged as a sequence of input vectors, we assume that a new vector is applied to the system inputs only when the system is in a steady state, *i.e.*, when the previous vector has been evaluated and the corresponding output values have been produced.

The behavioral descriptions we consider are coded in SystemC [23]. Each module in the system is an instance of the `SC_MODULE` class whose behavior is described through the `SC_METHOD` primitive.

## 4.4 High-level Test Generation

In this section we describe the high-level test generation environment we developed. In particular, Section 4.4.1 discusses the fault models adopted, and Section 4.4.2 presents the test generation algorithm we implemented.

### 4.4.1 High-level Fault Models

Several high-level fault models are available in the literature that can be used for assessing the goodness of test vectors while working at abstraction levels higher than the gate-level one. For the sake of this chapter, we considered the high-level fault models described in [12], which provide an accurate estimation of the test capabilities of input vectors while working on behavioral descriptions.

The fault models considered are:

- *Bit coverage*: each bit in every variable, signal or port in the model can be stuck to zero or one. The bit coverage measures the percentage of bit stuck-at that are propagated on the model outputs by a given test sequence.
- *Condition coverage*: each condition can be stuck-at true or stuck-at false. Then, the condition coverage is defined as the percentage of condition stuck-at that is propagated on the model outputs by a given test sequence.

In order to fruitfully exploits the aforementioned high-level fault models within a test generation tool, we developed a high-level fault simulation environment, which implements the *Saboteur* [2] approach through a two-step process:

1. The behavioral model under analysis is first instrumented by adding suitable statements that fulfill two purposes:
   - They alter the behavior of the model according to the supported fault models.
   - They allow observing the behavior of the model to gather meaningful statistics (in particular, they provide access to the contents of all the variables in the model).

   During this phase, the list of faults to be considered during fault simulation is computed and stored.
2. A given set of input vectors is applied to the inputs of the model by resorting to the simulation environment adopted. During the execution of the model, a preliminary simulation is performed without injecting faults, and the output trace of the model is recorded. Then, each fault in the previously computed fault list is injected and faulty output trace is recorded. By comparing the faulty trace with the fault-free one, we then compute the high-level coverage figure the vectors attain.

### 4.4.2 High-level Test Generation Algorithm

The test generation algorithm we developed is intended for refining an already existing set of test vectors, which can be either randomly generated or hand produced by designers.

For this purpose, our High-Level Test Generator (HLTG) implements a Random Mutation Hill Climber (RMHC) algorithm, whose pseudo-code is reported in the Figure 4.1.

An RMHC is a Hill Climber that, given a current solution, evaluates neighbor solutions in a completely random order until an improvement is found. When an improvement is found, the process is iterated over the new solution. The process is repeated until a given termination condition is not met.

In our algorithm, a solution is a sequence $S$ of test vectors; each test vector is applied over the model inputs according to the assumptions stated in Section 4.3.

Starting from an initial solution $S$, a new solution $S'$ is computed by applying a random mutation operator. This operator supports three types of mutation:

- It complements one randomly selected bit within a randomly selected vector of $S$.

- It increases the number of vectors in $S$ by adding a randomly generated vector in a randomly selected position in the test sequence.
- It decreases the number of vectors in $S$ by removing a randomly selected vector in the sequence.

The new solution $S'$is accepted if and only if it increases the goodness of the previous solution $S$.

```
HLTG($S$)
{
  while( termination condition not met )
  {
      $S'$ = apply_random_mutation($S$)
      if(Fitness($S'$) > Fitness($S$))
      {
        $S$ = $S'$
        if( new faults are detected )
          save_solution($S$)
      }
  }
}
```

**Figure 4.1.** The pseudo-code of the HLTG algorithm

In the HLTG algorithm, the goodness of a solution is defined as follows:

$$\texttt{Fitness}(S) = K_1\cdot\texttt{Coverage}(S) + K_2\cdot\mathrm{NS} \tag{4.1}$$

where:

- $\texttt{Coverage}(S)$ is the sum of the bit coverage and condition coverage as measured by the high-level fault simulator described in Section 4.4.1.
- NS is the number of different states the model traverses during the evaluation of a set of vectors. The state of the model is defined as the content of every variable in the model at the end of the evaluation of one input vector. This figure is computed by exploiting the information provided by the high-level fault simulator.

These two figures are linearly combined through two constants $K_1$ and $K_2$, whose values are selected to let the first term prevail over the second one. This assumption guarantees that a new solution is accepted only if it does not reduce the number of faults that the previous solution detects.

## 4.5 Testing Hardware/Software Systems

In this section we show how high-level generated vectors may be exploited for effectively testing hardware/software systems. In the following we consider hardware testing as the process of identifying manufacturing defects, while software testing is the process of identifying coding bugs.

The quality of vectors in testing hardware modules is measured as the number of stuck-at faults the vectors detect when the gate-level model of the hardware modules tested is fault simulated. The figure obtained through this process is called *gate-level stuck-at fault coverage*.

Conversely, the quality of vectors in testing software modules is measured as the number of software mutants the vectors detect; the figure obtained is called *mutant coverage*. The following mutants, which are a representative subset of possible coding bugs, were considered:

1. Replacement of arithmetic operators
2. Changing of the value of constants
3. Replacement of relational operators
4. Replacement of variables in operations and assignments
5. Replacement of logical operators
6. Deletion of operands from arithmetical operations.

In our analysis we considered five benchmarks coded in SystemC, whose characteristics are described in Table 4.1. Each benchmark is equipped with an initial set of vectors provided by the benchmark developer. The benchmarks come in part from the high-level synthesis'92 suite and in part from a set of in-house-developed models.

**Table 4.1.** Benchmark characteristics

| | Lines of SystemC code [#] | Variables [#] | Operations [#] | Inputs [#] | Outputs [#] |
|---|---|---|---|---|---|
| BARCODE | 119 | 5 | 4 | 4 | 4 |
| BIQUAD | 132 | 14 | 9 | 2 | 2 |
| DIFFEQ | 126 | 13 | 10 | 5 | 3 |
| ELLIPF | 81 | 29 | 26 | 8 | 8 |
| LRU | 91 | 13 | 4 | 1 | 1 |

We produced the hardware implementation of the benchmarks by synthesizing them through the SystemC compiler. Conversely, we produced the software implementation of the benchmarks by hand coding them in standard C language. Since the benchmarks are composed of one module only, they are implemented as a single task and thus we do not need to include any scheduler or any sort of operating system.

For comparison sake, we firstly created a hardware version of each benchmark and then computed a set of test vectors by exploiting a commercial gate-level test generation program (Synopsys `testgen`), whose results are reported in Section

4.5.1. Then, we generated a set of test vectors by exploiting a prototypical implementation of the HLTG tool, which amount to about 1000 lines of C code. We performed two types of test generation: one starting from random vectors (Section 4.5.2) and one starting from designer-supplied vectors (Section 4.5.3).

We then fault simulated the testgen and HLTG vectors for measuring their gate-level fault coverage, and we measured the number of mutants the vectors attain when the software implementation is considered.

All the experiments have been performed on Sun Enterprise 250 machines equipped with two processors running at 400 MHz and equipped with 2 Gb of RAM.

The experimental flows we exploited are shown in Figures 4.2 and 4.3. In our experiments, we adopted the Synopsys faultsim tool for gate-level stuck-at fault simulation, and the in-house-developed mSIM for simulating software mutants. We simulated the vectors computed by testgen and HLTG, resorting to the same tools and exploiting the same hardware/software descriptions; the obtained coverage figures are thus comparable.

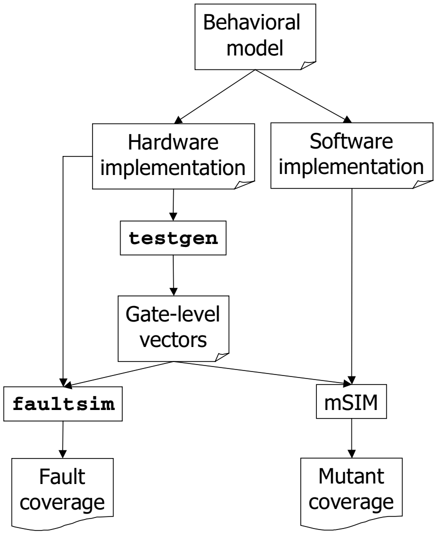


**Figure 4.2.** Simulation of testgen vectors

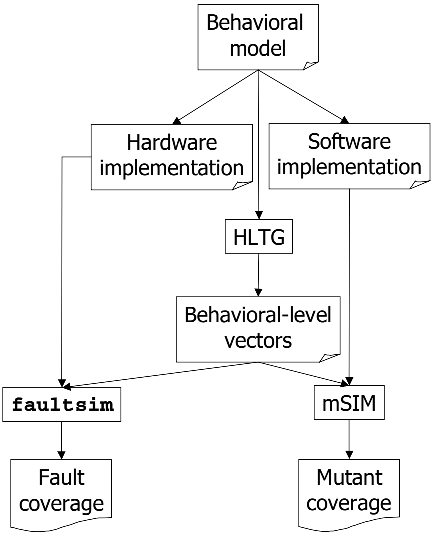


**Figure 4.3.** Simulation of HLTG vectors

## 4.5.1 testgen Results

In order to assess the goodness of the HLTG vectors, we performed a test generation experiments by exploiting the Synopsys testgen gate-level ATPG. The results obtained are shown in Table 4.2, where the gate-level stuck-at fault coverage (fault coverage for simplicity) and the mutant coverage are reported, as well as the test length and the time for running the ATPG.

**Table 4.2.** `testgen` results

| | Test length [#] | Time for running `testgen` [s] | Mutant coverage [%] | Fault coverage [%] |
|---|---|---|---|---|
| BARCODE | 2142 | 6430 | 100.0 | 68.7 |
| BIQUAD | 399 | 68,162 | 82.3 | 62.5 |
| DIFFEQ | 1162 | 14,901 | 100.0 | 97.4 |
| ELLIPF | 534 | 1736 | 100.0 | 98.3 |
| LRU | 397 | 22,146 | 84.9 | 65.2 |

### 4.5.2 Results Starting from Random Vectors

The results we gathered starting from randomly generated vectors are reported in Table 4.3. Columns *Initial* report the initial coverage, *i.e.*, the coverage we recorded for the randomly generated sequences before starting the HLTG tool. Conversely, columns *Final* report the final coverage we measured by simulating the vectors HLTG produced.

**Table 4.3.** HLTG results for random vectors

| | Test length [#] | | Time for running HLTG [s] | Mutant coverage [%] | | Fault coverage [%] | |
|---|---|---|---|---|---|---|---|
| | Initial | Final | | Initial | Final | Initial | Final |
| BARCODE | 25 | 10,372 | 955 | 68.9 | 100.0 | 46.7 | 68.7 |
| BIQUAD | 25 | 7,229 | 8959 | 47.9 | 100.0 | 30.8 | 85.6 |
| DIFFEQ | 25 | 6,579 | 3979 | 98.1 | 98.1 | 54.9 | 98.5 |
| ELLIPF | 25 | 5,435 | 439 | 100.0 | 100.0 | 95.1 | 97.7 |
| LRU | 25 | 7,152 | 1213 | 31.5 | 100.0 | 1.7 | 73.7 |

These results show the effectiveness of the HLTG tool, which is able to produce high-quality vectors by exploiting just the knowledge of the behavior of the benchmark considered. For all the benchmarks, both mutant coverage and fault coverage are always greater (or at least equal to) that obtained by vectors generated by `testgen`.

### 4.5.3 Results Starting from Designer Vectors

The results we gathered starting from already-existing vectors are reported in Table 4.4.

**Table 4.4.** HLTG results for designer vectors

| | Test length [#] | | Time for running HLTG [s] | Mutant coverage [%] | | Fault coverage [%] | |
|---|---|---|---|---|---|---|---|
| | Initial | Final | | Initial | Final | Initial | Final |
| BARCODE | 700 | 7902 | 463 | 100.0 | 100.0 | 74.7 | 79.4 |
| BIQUAD | 200 | 10,348 | 7401 | 100.0 | 100.0 | 80.9 | 83.4 |
| DIFFEQ | 100 | 6162 | 4923 | 100.0 | 100.0 | 80.4 | 98.5 |
| ELLIPF | 25 | 5435 | 439 | 100.0 | 100.0 | 94.1 | 97.7 |
| LRU | 200 | 5837 | 353 | 100.0 | 100.0 | 60.2 | 75.6 |

For the sake of these experiments, we exploited the vectors the benchmark designer developed for validation purposes. These are vectors the designer used to identify bugs possibly introduced in the benchmark behavior during its design stage.

These results show that, although the designer already produced high-quality vectors, HLTG is able to improve them further.

### 4.5.4 Result Discussion

By comparing the figures in Tables 4.2, 4.3, and 4.4 we can observe that:

- The number of vectors HLTG produced is usually higher than that coming from the gate-level ATPG. This result is not surprising, due to the limited complexity of the high-level algorithm, where few efforts are devoted to reducing the test length.
- For all the benchmarks considered, HLTG is able to finish the test generation procedure in an amount of time 3 to 60 times lower than that spent by `testgen`. By addressing the test on behavioral descriptions, and thus neglecting all the details that gate-level models imply, we can effectively reduce the test generation time.
- Although very simple, the HLTG is able to produce high-quality test vectors. The coverage results that the HLTG vectors provide are indeed comparable to that of vectors generated by a commercial gate-level ATPG, and in most cases better, as far as both mutant and stuck-at fault coverage are considered.

## 4.6 Validating Application-specific Processors

In this section we show how high-level generated vectors may be exploited for effectively validating processor cores that have been customized for being deployed in embedded applications.

Thanks to the availability of deep sub-micron technologies, designers have now plenty of silicon area for their designs, up to the point that it is now common to find embedded systems integrated on a single chip that features memory modules, processor cores and even embedded programmable logic modules. To support the design of such kinds of system effectively, which are known as SOCs, vendors started offering Intellectual Property- (IP-) cores ready for implementing complex tasks; for example, processor cores are now available ranging from simple controllers (like those implementing the Intel 8051 instruction set) to more complex pipelined processors (like those based on the SPARC v8 architecture). Now that such IP-cores are available, most of the work in SOC design consists in integrating different IP-cores: designers can indeed implement an SOC by properly customizing and connecting the IP-cores needed coming from potentially very different sources. To foster this design approach, several attempts have been made to provide IP-cores with standard interfaces. The idea behind them is to simplify the communications between heterogeneous IPs during both normal operations (like for example the WISHBONE interface [20]) and test operations (like the IEEE P1500 standard).

Although silicon area is available in quantity, designers still face the need to minimize the size of their designs. Large-area occupation still has several drawbacks: high manufacturing costs, low yield, high power consumption, just to mention a few of them. In an IP-core-based design flow, this implies the possibility of customizing the IPs adopted to make them implement just the features needed. IP customization is particularly efficient when the SOC is intended for deployment in an embedded application. In this case all the IP-cores the SOC employs perform a very specific task that does not change during the SOC lifetime, and which is well defined from the beginning of the SOC design. Moreover, the customization is effective only when very complex IP-cores are considered, as in the case of processors. For example, if the SOC designed is using a processor IP whose architecture embeds a parallel multiplier, but the embedded application it is aimed at does not require multiplication operations, designers can save a huge amount of area by removing the parallel multiplier from the IP.

Vendors must implement processor customization when IPs come under the form of hard cores. In this case, designers cannot modify the IP architecture and thus they have to rely fully on their suppliers. This may have a dramatic impact on the IP cost, but it also greatly simplifies the task for the IP end users, which are freed from the very complex task of guaranteeing IP correctness. It is indeed up to IP suppliers to identify possible bugs introduced in the core during the customization process.

Conversely, when processor cores are available as soft IPs, *i.e.*, designers have full access to the IP source code, the process of core customization may be performed by the end user, who also becomes responsible for guaranteeing the correct operation of the IP, *i.e.*, of the *validation* of the resulting IP, after its unused components have been removed.

The problem of validating processor cores may be tackled either by means of formal methods or by means of simulation-based techniques. Formal methods have been successfully applied even to very complex architectures [13,24,25], but their limitations often make them suitable only for validating single components. As a result, most of the validation effort is demanded by simulation-based techniques: the processor is stimulated extensively with a wide range of workloads, whose aim is to cover all the possible corner cases, possibly highlighting design errors. To achieve such a goal successfully, the process of generating the workload is crucial, since from its goodness it depends the possibility of effectively discovering design bugs. Several approaches have been proposed in the literature to solve the complex problem of generating suitable workloads for general-purpose processors. They rely either on high-level behavioral HDL descriptions of the processors, or on more detailed RT models to generate and evaluate the goodness of the computed workloads versus predefined metrics [4,5]. No matter which approach is adopted, the workloads produced consist of *test programs* the processor core should execute and suitable *input stimuli* for the test programs.

The workload computation problem can be greatly simplified when the processor cores to be validated are intended for being deployed in embedded applications. In this case the program that a core should run is *known in advance*, while for general-purpose processors the executed program may change from time to time. The processor is indeed intended for running just one application and, therefore, the test program coincides with the embedded application. Designers should thus focus their efforts in the development of the input stimuli for the test program, only.

This observation suggests that high-level generated vectors that extensively test the behavioral model of an embedded application may also be successfully adopted for validating the processor core that has been customized for running the application.

To validate this novel idea, we developed an automatic flow that, starting from the high-level model of the embedded application and a description of the processor core devoted to its execution, performs the following operations:

- It automatically customizes the selected processor by removing from its description all the unused instructions, so that the resulting core embeds only those hardware components that are actually needed by the embedded application.
- It automatically generates a set of input stimuli suitable for being used during the validation of the processor core obtained.

Conversely, from the already available approach, input stimuli are generated while analyzing *only* the high-level model of the application the processor executes, while all the details about the underlying hardware, *i.e.*, the processor, are neglected.

After describing the design flow we developed, we report some experimental results we gathered on a soft core implementing the Intel 8051 instruction set, showing the effectiveness of the approach we propose.

### 4.6.1 Design Flow

In this section we describe the approach we developed for customizing a given processor core and generating suitable validation inputs. In developing our approach, we assumed that the embedded application considered is composed of three phases:

- An *acquisition* phase, during which the data the application is intended to process are read from input devices.
- A *processing* phase, during which the acquired data are elaborate by the algorithms the embedded application implements.
- A *presentation* phase, during which the results obtained are sent to output devices.

These phases can be intermingled; indeed, our approach does not mandate that one phase is completed before another one is started.

We also assumed that the processor core targeted is available as a synthesizable RT model. The processor source code is not encrypted and all its details are accessible to designers. Finally, we assumed that a simulation-based approach is exploited for performing the validation of the considered processor.

The flow we developed under the aforementioned assumptions is shown in Figure 4.4. Three main phases compose our flow. The *Instruction Set Extraction* flow is shown in the leftmost part of Figure 4.4. According to this, the embedded-application source code is first compiled and then linked with the libraries needed, thus obtaining the binary code the processor core should execute. The source code is obtained starting from the application high-level model by translating it from SystemC to the C language as we already did in Section 4.5. Then, a binary code analyzer tool identifies the subset of the processor instruction set that is needed for executing the given embedded application. The result of the Instruction Set Extraction flow is thus the list of assembly instructions the processor should implement in order to execute correctly the embedded application it is devoted to.

The information obtained is then forwarded to the *Processor Configuration* flow (shown in the center of Figure 4.4), which takes care of generating the proper processor model implementing only the required subset of the processor instruction set. A previously defined processor core database is exploited during this step, which contains for each instruction in the processor instruction set, the list of VHDL statements needed for its decoding, sequencing and execution. The database is exploited by the processor configurator tool for generating the VHDL source code of all the modules in the processor that are devoted to instruction management, namely: decoding unit, control unit, arithmetic/logic unit. At the end of the Processor Configuration flow an instance of the adopted processor core is available that has been customized for executing the given embedded application.

The last phase is the *Input Generation* flow, which is depicted in the rightmost part of Figure 4.4. According to this flow, the embedded application high-level model is processed by the HLTG tool described in Section 4.4.2, which is in charge of computing a set of input stimuli, *i.e.*, *test vectors*. While performing the simulations needed for validating the processor, the test vectors generated are provided to the embedded application as inputs for its acquisition phase. At the end of this flow, a set of validation input stimuli is available that designers may use to prove the correctness of the customized processor core obtained while running the given embedded application.

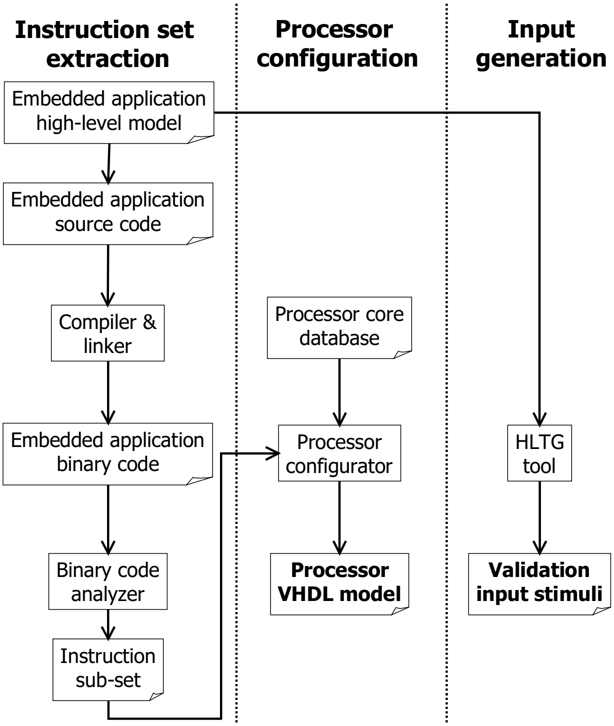


**Figure 4.4.** The proposed processor customization and validation flow

### 4.6.2 Experimental Results

In this section we report the results coming from several experiments we performed to analyze the capabilities of the design flow described in Section 4.6.1. Section 4.6.3 reports results concerning our processor-customization approach, which show that an instruction set may be significantly pruned when a given embedded application is concerned. Section 4.6.4 reports results about the test vectors generation approach we developed. These results confirm the soundness of the proposed approach, and they show that the high-level fault models adopted, which are applied only to the application running on the processor and that we used to drive the test generation algorithm, are in very good agreement with lower level metrics measured on the processor hardware description.

### 4.6.3 Results of the Processor Customization

The processor we selected for developing some benchmark applications is a soft core that implements the Intel 8051 instruction set. The soft core is coded in about

7200 lines of synthesizable VHDL, and its instruction set implements 106 instructions.

As a preliminary step for the application of our design flow, we manually inspected the VHDL code of the processor, and we identified all the information needed by the binary code analyzer tool and for building the processor core database depicted in Figure 4.4. This step lasted for about 3 days and was performed by a skilled VHDL designer. It is worthwhile underlining that, although expensive, this step needs to be performed only once, whenever a new processor core is introduced in the design flow. The information obtained can then be re-used for any new embedded application exploiting the core already analyzed.

**Table 4.5.** The applications considered

| Application name | Lines of C code [#] | Instruction set [#] | Validation input stimuli [#] | CPU time [s] |
|---|---|---|---|---|
| BARCODE | 198 | 27 | 4731 | 955 |
| ELLIPF | 113 | 19 | 130 | 439 |
| LRU | 107 | 36 | 6810 | 1213 |

After this preliminary step, we considered the three applications summarized in Table 4.5, taken from the high-level synthesis’92 suite. For each of them, we applied the design flow in Figure 4.4, thus generating a customized processor core, and the corresponding validation input stimuli. The binary code of each application was obtained through the KEIL C compiler [26].

Table 4.5 reports for each application the number of C lines in its source-level code. Moreover, it reports the number of instructions within the Intel 8051 instruction sets that are needed for running it and that are implemented by the customized version of the processor obtained. Finally, it reports the number of test vectors in the validation input stimuli set HLTG computed, as well as the CPU time for processor customization and HLTG execution. The figures in Table 4.5 show the relevancy of the processor customization approach we adopted: for the applications considered, just a relatively low number of instructions are needed among the whole Intel 8051 instruction set. By pruning the initial instruction set of the unused instructions we can thus significantly reduce the area occupation of the processor core synthesized.

### 4.6.4 Results of the Test Vector Generation

We adopted the number of processor instructions that have been fully tested by *S* as a measure of the goodness of a given set of validation input stimuli *S*, obtaining the figure we called *instruction coverage*. An instruction is considered *tested* when simulating *S* all the statements and the branches belonging to the VHDL implementation of the instruction are executed.

**Table 4.6.** Coverage results for HLTG and randomly generated validation input stimuli

| Application name | Instruction set [#] | Instructions tested by HLTG vectors [%] | Instructions tested by random vectors [%] |
|---|---|---|---|
| BARCODE | 27 | 100.0 | 96.2 |
| ELLIPF | 19 | 100.0 | 100.0 |
| LRU | 36 | 100.0 | 97.2 |

To measure the instruction coverage, we applied a given set of input stimuli to the application by simulating[1] the execution of its binary code on the VHDL model of the customized processor. In order to implement the input/output communications needed by the acquisition and presentation phases described in Section 4.6.1, we resorted to the four input/output ports the Intel 8051 offers. They are 8-bit-wide bi-directional ports, which are memory mapped in the Intel 8051 addressing space. From the application developer point of view, the input/output ports correspond to C variables (`P0`, `P1`, `P2` and `P3`) that can be either read or written. Read operations on one of these variables correspond to data transfers from input devices to the processor, while write operations correspond to data transfers from the processor to output devices.

Two sets of input stimuli have been used. The first set is composed of the vectors computed by HLTG, and the second set is composed of randomly generated vectors. In both the experiments the same number of test vectors have been simulated.

From the results reported in Table 4.6, we can observe that HLTG vectors are able to test all the instructions successfully in the processor cores considered, while random vectors fail short in achieving such a goal for two of the three applications considered. In these cases, random vectors were not able to cover part of the VHDL statements implementing the `JNZ` instruction (for the LRU application) and `JNE` instruction (for the BARCODE application).

These results suggest the importance of cleverly selecting the test vectors to be used during simulation-based validation. Although simple, the test vector generation approach adopted is rapidly able to provide useful test vectors that overcome the limitations of purely randomly generated vectors.

To investigate better the capabilities of the high-level fault models described in Section 4.4 to describe low-level errors accurately, in Table 4.7 we compared the coverage figures (as defined in Section 4.4.2) measured on the high-level model of the applications considered with those reported in Table 4.6, which have been measured by simulating the VHDL model of the processor.

[1] Simulations have been performed through the ModelSim VHDL simulator on a Sun Enterprise/250 machine running at 400 MHz and equipped with 2 Gbytes or RAM.

**Table 4.7.** Comparing high-level with low-level metrics

| Application name | High-level coverage of HLTG vectors [%] | High-level coverage of random vectors [%] | Instructions coverage of HLTG vectors [%] | Instructions coverage of random vectors [%] |
|---|---|---|---|---|
| BARCODE | 97.2 | 75.6 | 100.0 | 96.2 |
| ELLIPF | 99.9 | 99.9 | 100.0 | 100.0 |
| LRU | 98.8 | 98.2 | 100.0 | 97.2 |

As the reader can observe from Table 4.7, the high-level coverage has the same trend as the instruction coverage. For both BARCODE and LRU, HLTG vectors perform better than random ones when coverage figures are measured both at the high level (on the application source code) and at the low level (on the processor VHDL model). Similarly, random vectors and HLTG vectors provide the same figures for ELLIPF when evaluated either on the application source code or on the processor model. These results indicate that accurate testability estimation can be performed starting from purely behavioral descriptions (such as in the case of the source-level code of an application) while all the details of the underlying hardware (the processor devoted to executing the application) are neglected.

## 4.7 Conclusions

This chapter presents an approach to high-level test generation suitable for providing designers with high-quality vectors at low cost. By exploiting a heuristic algorithm and by working only on the purely behavioral system descriptions, our approach is able to provide high-quality vectors, or to improve already existing ones, making them suitable for different purposes. In this chapter we reported experimental evidence of goodness of high-level generated vectors in testing both the hardware and the software modules of complex systems, as well as their capability of validating processor cores that have been customized for running a given embedded application.

High-level generated vectors are not the ultimate solution for the test and validation problem; indeed, they are not able to cover all the possible faults in a system, mostly due to the existence of faults that can be hardly modeled at abstraction levels higher than the gate-level one.

Nevertheless, high-level generated vectors can be of great help in reducing test generation costs thanks to the exploitation of abstract models that require less CPU resources for simulation and test generation. Moreover, high-level generated vectors can be used for both testing the hardware components and for validating the software components.

## References

[1] Beizer B (1990) Software testing techniques. Van Nostrand Rheinold, New York

[2] Boué J, Pétillon P, Crouzer Y (1998) MEFISTO-L: a VHDL-based fault injection tool for the experimental assessment of fault tolerance. In: Int'l Symposium on Fault Tolerant Computing, 168-173

[3] Chiusano S, Corno F, Prinetto P (1999) Exploiting behavioral information in gate level ATPG. The Journal of Electronic Testing: Theory and Applications, Kluwer Academic Publishers, (14): 141-148

[4] Corno F, Sonza Reorda M, Squillero G, Violante M (2001) On the test of microprocessor IP cores, In: IEEE Design, Automation & Test in Europe, 209-213

[5] Corno F, Cumani G, Sonza Reorda M, Squillero G (2003) Fully automatic test program generation for microprocessor cores. In: IEEE Design, Automation & Test in Europe, 1006-1011

[6] Corno F, Sonza Reorda M, Squillero G (2000) High level observability for effective high level ATPG. In: 18th IEEE VLSI Test Symposium, 411-416

[7] Coward PD (1990) Symbolic execution and testing. In: IEE Colloquium on Software Testing for Critical Systems, 2/1-2/3

[8] DeMillo RA, Guindi DS, McCracken WM, Offutt AJ, King KN (1988) An extended overview of the Mothra software testing environment. In: IEEE Workshop on Software Testing, Verification, and Analysis, 142-151

[9] Edwards S, Lavagno L, Lee EA, Sangiovanni Vincentelli A (1997) Design of embedded systems: formal models, validation, and synthesis. Proceedings of IEEE, 85(3): 366-390

[10] Fallah F, Ashar P, Devadas S (1999) Simulation vector generation from HDL descriptions for observability-enhanced statement coverage. In: Design Automation Conference, 666-671

[11] Ferrandi F, Fummi F, Sciuto D (1998) Implicit test generation for behavioral VHDL Models. In: IEEE Int'l Test Conference, 587-596

[12] Ferrandi F, Fummi F, Sciuto D (2002) Test generation and testability alternatives exploration of critical algorithms for embedded applications. IEEE Transactions on Computers, 51(2): 200-215

[13] Harman NA, Verifying a simple pipelined microprocessor using MAUDE, In: Lecture Notes in Computer Science, 2001, vol. 2267, 128-142

[14] Howden WE (1976) Reliability of the path analysis testing strategy. IEEE Transactions on Software Engineering, 2(3): 208-215

[15] Howden WE (1978) Functional program testing. In: IEEE Int'l Conference on Software and Applications, 321-325

[16] Jervan G, Peng Z, Goloubeva O, Sonza Reorda M, Violante M (2002) High-level and hierarchical test sequence generation. In: IEEE Int'l Workshop on High Level Design Validation and Test, 169-174

[17] King JS (1976) Symbolic execution and program testing. Communications of the ACM, (19): 385-394

[18] Moundanos D, Abraham JA, Hoskote Y (1996) A unified framework for design validation and manifacturing test. In: IEEE Int'l Test Conference, 875-884

[19] Offutt AJ (1994) A practical system for mutation testing: help for the common programmer. In: IEEE Int'l Test Conference, 1994, 824-830

[20] OPENCORES, WISHBONE system-on-chip (SOC) interconnection architecture for portable IP cores, Revision B.3, September 2002

[21] Rudnick EM, Vietti R, Ellis A, Corno F, Prinetto P, Sonza Reorda M (1998) Fast sequential circuit test generation using high level and gate level techniques. In: IEEE European Design Automation and Test Conference, 570-576

[22] Santos MB, Goncalves FM, Teixeira IC, Teixeira JP (2000) RTL-based functional test generation for high defects coverage in digital SOCs. In: IEEE European Test Workshop, 99-104

[23] SystemC User's Guide, Synopsys, CoWare, Frontier Design

[24] Van Campenhout D, Mudge TN, Hayes JP, High-level test generation for design verification of pipelined microprocessors, In: Design Automation Conference, 1999, 185-188

[25] Velev MN, Bryant RE, Formal verification of superscalar microprocessors with multicycle functional units, Exception, And Branch Prediction, In: Design Automation Conference, 2000, 112-117

[26] www.keil.com

# 5 Test Generation: A Hierarchical Approach

G. Jervan, R. Ubar, Z. Peng, P. Eles

Linköping University, Linköping, Sweden
Tallinn University of Technology, Tallin, Estonia

## 5.1 Abstract

Advances in design tools and methods have led to an increasing amount of design activities being performed at higher levels of abstraction. Testability, on the other hand, is usually considered only when the detailed structural information of the design is available. This is mainly due to the lack of general applicability of the existing high-level test generation and design-for-test methods. In this chapter we will present an improvement of the classical hierarchical test generation approach by extending it to the higher levels of abstraction, while still considering the structural information from the lower levels. The approach proposed makes successful use of both high-level fault models and the classical gate-level fault models, and obtains results that are better than those obtained by a pure high-level test generator.

## 5.2 Introduction

As described in the previous chapters, the introduction of System-on-Chip (SOC) entails several challenges with respect to the design, test and manufacturing of such systems. To cope with the challenges faced by SOC designers, tools and techniques dealing with design at higher levels of abstraction have been developed. For example, behavioral-level synthesis tools and hardware/software co-design techniques are starting to play an important role in the initial phases of the design process. The main advantage of deploying such high-level design tools is the possibility to evaluate quickly the costs and benefits of different architecture alternatives, including both hardware and software components, starting from a high-level functional specification of the system implemented.

While the main design focus is quickly moving toward higher levels of abstraction, the test issues are still considered only when a detailed description of the design is available, typically at the gate level for test sequence generation and at register transfer (RT) level for design for testability structure insertion.

To address the problems associated with test generation and design-for-test (DfT), when performed at the later design stages, intensive research efforts have been devoted to devise solutions to test sequence generation and DfT in the early design phases, mainly at the RT level. For high-level test generation, several proposed approaches are able to generate test patterns of good quality, sometimes even better than those of gate-level Automatic Test Pattern Generation (ATPG) tools. However, owing to the lack of general applicability, most of these approaches are still not used in the industry.

This chapter presents a high-level hierarchical test generation (HTG) approach for improving the results obtained by a pure high-level test generator. The hierarchical test generator takes into account structural information from lower levels of abstraction while generating test sequences on the behavioral level. We will start our discussion with the description of the modeling technique we use to model the design under test and the corresponding fault modeling techniques. Then later in this chapter the HTG approach will be described.

## 5.3 Modeling with Decision Diagrams

Test generation for digital systems encompasses three main activities: selecting a description method, developing a fault model, and generating tests to detect the faults covered by the fault model. The efficiency of test generation (quality and speed) is highly dependent on the description method and fault models which have been chosen. In order to generate tests at the high abstraction levels, we need a modeling technique that can capture designs at the levels in concern. Since the HTG approach takes advantages of both high-level and low-level design information, we need a modeling technique which spans several levels of abstraction. This section will describe such a model, called Decision Diagrams (DDs).

### 5.3.1 State of the Art

For high-level test generation, different high-level design and fault models have been introduced. The main idea of high-level modeling is to capture the high-level description of the system in a formal model, and to obtain different incorrect versions of the design by introducing a fault into the model. This approach is called model perturbation [6]. The models can be “perturbed” in certain ways, *e.g.*, by truth-table modification, micro-operation modification, *etc.* In one way or the other, this idea is implemented in different high-level fault models for different classes of digital systems.

In the case of microprocessors, individual functional fault models and their corresponding test strategies have been developed for different function classes, such as register decoding, instruction decoding, control, data storage, data transfer, data manipulation, *etc.* [2, 14]. The main disadvantage of this approach is that only microprocessors are handled and the results obtained cannot be extended to cover the general digital systems testing problem. When using Register Transfer Level

(RTL) languages, a formal definition of an RTL statement is defined, and nine categories of functional faults for components of RTL statements are identified [12, 13]. Recently, a lot of attention has been devoted to generating tests directly from high-level description languages [4, 5, 18]. Some attempts to develop special functional fault models for different data-flow network units like decoders, multiplexers, memories, Programmable Logic Aarrays (PLAs), *etc.* are described in [1].

The drawback of traditional multi-level and hierarchical approaches to test generation lies in the need of different languages and models for different levels. For example, one might use logic expressions for combinational circuits; state transition diagrams for Finite-State Machines (FSMs); abstract execution graphs, system graphs, instruction-set architecture descriptions, flow-charts, hardware description languages, or Petri nets for system-level description, *etc*. All these models need different manipulation algorithms and fault models which are difficult to merge into a coherent hierarchical test method. To address this problem, DDs can be used [3, 9, 11, 15, 16, 17]. Binary DDs (BDDs) have already found very broad applications in logic design and in logic testing [3, 9]. Structurally Synthesized BDDs (SSBDDs) are able to represent gate-level structural faults directly in the graph [15, 16]. Recent research has shown that generalization of BDDs for higher levels provides a uniform model for both gate- and RT-level [11, 17], and even behavioral-level test generation [7, 8].

In our approach, a method for describing digital systems and for modeling faults is based on DDs. DDs serve as a basis for a general theory of test design for mixed-level representations of systems, in a manner similar to the Boolean algebra for the plain logical level. DDs can be used to represent systems uniformly either at the logic level, high level or simultaneously at both levels. The fault model defined on DDs represents a generalization of the classical gate-level stuck-at fault model.

### 5.3.2 Modeling Digital Systems by Binary Decision Diagrams

Let us first consider BDDs in order to illustrate the basic notations. BDDs are a special case of DDs that are described later in this chapter for behavior-level diagnostic modeling of digital systems. We will first describe logic-level BDDs to prepare the readers for a better understanding of the generalization of BDDs for higher level system representation. We will use the graph-theoretical definitions instead of traditional logic-oriented *ite* expressions [3, 9], because all the procedures defined further for DDs are based on the topological reasoning rather than on graph symbolic manipulations as in the case of BDDs.

***Definition 5.1:*** A BDD that represents a Boolean function $y = f(X)$, $X = (x_1, x_2, ..., x_n)$, is a directed acyclic graph $G_y = (M, \Gamma, X)$, with a set of nodes $M$ and a mapping $\Gamma$ from $M$ to $M$. $M = M^N \cup M^T$ consists of two types of node: nonterminal $M^N$ and terminal $M^T$ nodes. A terminal node $m^T \in M^T = \{m^{T,0}, m^{T,1}\}$ is labeled by a constant $e \in \{0, 1\}$ and is called a *leaf*; all nonterminal nodes $m \in M^N$ are labeled by variables $x \in X$, and have exactly two successors. Let us denote the variable associated with node $m$ as $x(m)$, then $m^0$ is the successor of $m$ for the value $x(m) = 0$ and $m^1$ is the successor of $m$ for $x(m) = 1$.

***Definition 5.2:*** By the value of $x(m) = e$, $e \in \{0, 1\}$, we say the edge between nodes $m \in M$ and $m^e \in M$ is *activated.* Consider a situation where all the variables $x \in X$ are assigned by a Boolean vector $X^t \in \{0, 1\}^n$ to some value. The activated edges by $X^t$ form an *activated path* $l(m_0, m^T) \subseteq M$ from the root node $m_0$ to one of the terminal nodes $m^T \in M^T$.

***Definition 5.3:*** We say that a BDD $G_y = (M, \Gamma, X)$ represents a Boolean function $y = f(X)$ iff for all the possible vectors $X^t \in \{0, 1\}^n$ a path $l(m_0, m^T) \subseteq M$ is activated so that $y = f(X^t) = x(m^T)$.

***Definition 5.4:*** Consider a BDD $G_y = (M, \Gamma, X)$ where $X$ is the vector of literals of a function $y = P(X)$ represented in the equivalent parenthesis form [16], the nodes $m \in M^N$ are labeled by $x(m)$ where $x \in X$ and $|M| = |X|$. The BDD is called an SSBDD iff there exists a one-to-one correspondence between literals $x \in X$ and nodes $m \in M^N$ given by the set of labels $\{x(m) \mid x \in X, m \in M^N\}$, and for all the possible vectors $X^t \in \{0,1\}^n$ a path $l(m_0, m^T)$ is activated, so that $y = f(X^t) = x(m^T)$.

Unlike the traditional BDDs [3, 9], SSBDDs [16] support structural representation of gate-level circuits in terms of signal paths. By superposition of DDs [16], we can create SSBDDs with one-to-one correspondence between graph nodes and signal paths in the circuit. The whole circuit can then be represented as a network of tree-like subcircuits (macros), each of them represented by an SSBDD. Using SSBDDs, it is possible to ascend from the gate level to a higher macro level without losing accuracy in representing gate-level signal paths.

Our intention is to make use of the SSBDDs to capture both the structural and functional properties of a given circuit in order to generate high-quality test patterns.

Figure 5.1 shows a representation of a tree-like combinational circuit by an SSBDD. For simplicity, values of variables on edges of the SSBDD are omitted (by convention, an edge going to the right corresponds to 1, and an edge going down corresponds to 0). Also, terminal nodes with constants 0 and 1 are omitted: leaving the graph to the right corresponds to $y = 1$, and down to $y = 0$. The SSBDD graph contains seven nodes, and each of them represents a signal path in the given subcircuit (denoted as a macro in Figure 5.1). An activated path in the graph corresponding to the input pattern $x_1x_2x_3x_4x_5x_6 = 110100$ is highlighted in bold. The value of the function $y = 1$ for this pattern is determined by the value of the variable $x_5 = 1$ in the terminal node of the path.

***Procedure 1:*** Test generation. To generate a test for a node $m \in M^N$ in $G_y$, the following paths have to be activated:

(1) $l(m_0, m)$,
(2) $l(m^1, m^{T,1})$, and
(3) $l(m^0, m^{T,0})$.

To generate a test pattern for the path from $x_{7,1}$ to $y$ in the circuit by using SSBDDs means to generate a test pattern for the corresponding node $x_{7,1}$ in the graph. To test the node $x_{7,1}$, according to Procedure 1, the following paths should be activated; $(6, \neg 1, 2, 7_1)$, $(\neg 1, m^{T,1})$, and $(\neg 1, m^{T,0})$, which produces the test pattern: $x_1x_2x_3x_4x_5x_6$ = 11xx00. For example, to test a physical defect of a bridge between the lines 6 and 7, which is activated on the line 7, additional constraint $W = \neg x_6 \wedge x_7 = 1$ has to be used, which updates the test vector to 111x00.

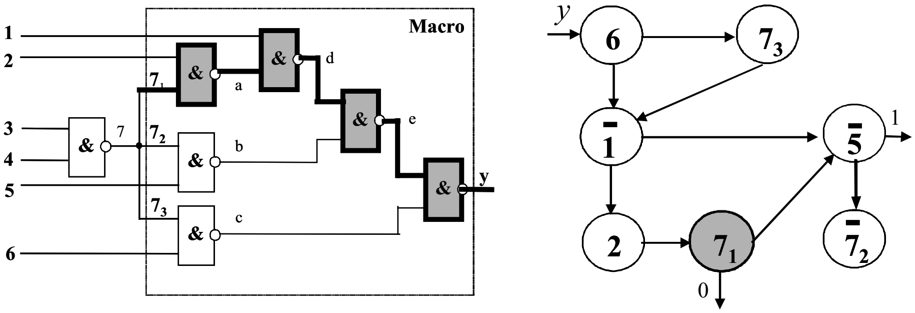


**Figure 5.1.** A gate-level circuit and its corresponding SSBDD

### 5.3.3 Modeling with a Single Decision Diagram on Higher Levels

Consider now a digital system $S = (Z, F)$ as a network of components (or processes), where $Z$ is the set of variables (Boolean, Boolean vectors or integers) that represent connections between components, as well as inputs and outputs of the network. Denote by $X \subset Z$ and $Y \subset Z$, correspondingly, the subsets of input and output variables. $V(z)$ denotes the set of possible values for $z \in Z$, which are finite.

Let $F$ be the set of digital functions on $Z$: $z_k = f_k(z_{k,1}, z_{k,2}, \dots, z_{k,p}) = f_k(Z_k)$ where $z_k \in Z$, $f_k \in F$, and $Z_k \subset Z$. Some of the functions $f_k \in F$, for the state variables $z \in Z_{\mathrm{STATE}} \subset Z$, are next state functions.

***Definition 5.5:*** A *decision diagram* is a directed acyclic graph $G = (M, \Gamma, Z)$ where $M$ is a set of nodes, $\Gamma$ is a relation in $M$, and $\Gamma(m) \subset M$ denotes the set of successor nodes of $m \in M$. The nodes $m \in M$ are marked by labels $z(m)$. The labels can be variables $z \in Z$, algebraic expressions $f_m(Z(m))$ of $Z(m) \subseteq Z$, or constants.

For nonterminal nodes $m \in M^N$, where $\Gamma(m) \neq \emptyset$, an onto function exists between the values of $z(m)$ and the successors $m^e \in \Gamma(m)$ of $m$. By $m^e$ we denote the successor of $m$ for the value $z(m) = e$. The edge $(m, m^e)$ which connects nodes $m$ and $m^e$ is called *activated* iff there exists an assignment $z(m) = e$. Activated edges, which connect $m_i$ and $m_j$, make up an *activated path* $l(m_i, m_j) \subseteq M$. An activated path $l(m^0, m^T) \subseteq M$ from the initial node $m^0$ to a terminal node $m^T$ is called a *full activated path*.

***Definition 5.6:*** A decision diagram $G_{z,k}$ represents a high-level function $z_k = f_k\,(z_{k,1}, z_{k,2}, \ldots, z_{k,p}) = f_k\,(Z_k)$, $z_k \in Z$ iff for each value $v(Z_k) = v(z_{k,1}) \times v(z_{k,2}) \times \ldots \times v(z_{k,p})$, a full path in $G_{z,k}$ to a terminal node $m^T \in M^T$ in $G_{z,k}$ is activated, so that $z_k = z(m^T)$ is valid.

Depending on the class of the system (or its representation level), we may have various DDs, where nodes have different interpretations and relationships to the system structure. In RTL descriptions, we usually partition the system into control and data parts. Nonterminal nodes in DDs correspond to the control path, and they are labeled by state and output variables of the control part serving as addresses or control words. Terminal nodes in DDs correspond to the data path, and they are labeled by the data words or functions of data words, which correspond to buses, registers, or data manipulation blocks.

When using DDs for describing complex digital systems, we first have to represent the system by a suitable set of interconnected components (combinational or sequential subcircuits). Then, we have to describe these components by their corresponding functions, which can be represented by DDs.

```
if (y1=0)
  c:=R1+R2;
else
  c:=IN+R2;
endif;

if (y2=0)
  d:=R1*R2;
else
  d:=IN*R2;
endif;

case y3
  0: e:=c;
  1: e:=IN;
  2: e:=R1;
  3: e:=d;
end case;
if (y4=2)
  R2:=e;
```

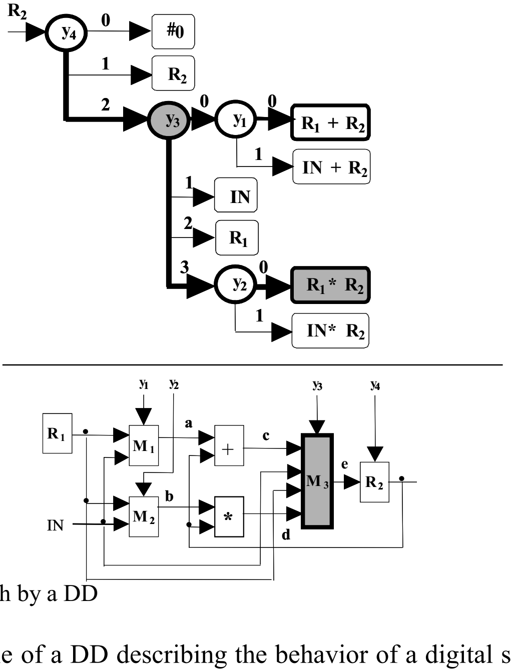


**Figure 5.2.** Representing a data path by a DD

Figure 5.2 depicts an example of a DD describing the behavior of a digital system together with its possible RT-level implementation. The variables $R_1$, $R_2$ and $R_3$ represent registers, IN represents the input bus, the integer variables $y_1$, $y_2$, $y_3$, $y_4$ represent the control signals, $M_1$, $M_2$, $M_3$ are multiplexers, and the functions $R_1$+$R_2$ and $R_1$*$R_2$ represent the adder and multiplier respectively. Each node in the

DD represents a subcircuit of the system (*e.g.* the nodes $y_1$, $y_2$, $y_3$, $y_4$ represent multiplexers and decoders). The whole DD describes the behavior of the input logic of the register $R_2$. To test a node means to test the corresponding subcircuit.

For test pattern simulation, a path is traced in the graph, guided by the values of input variables until a terminal node is reached, similar to the case of SSBDDs. In Figure 5.2 the result of simulating the vector $y_1$, $y_2$, $y_3$, $y_4$, $R_1$, $R_2$, IN = 0,0,3,2,10,6,12 is $R_2 = R_1 * R_2 = 60$ (bold arrows mark the activated path). Instead of simulating all the components in the circuit by a traditional approach, in the DD only three control variables are visited during simulation, and only a single data manipulation $R_2 = R_1 * R_2$ is carried out.

## 5.4 Hierarchical Test Generation with Decision Diagrams

One possible approach to deal with test generation complexity is to raise up the level of design abstractions at which the basic test generation procedure is performed. In the following we will describe an approach that performs the test generation procedure using the high-level behavioral description captured by DDs, but at the same time takes into account some detailed information of the basic components at the lower levels.

At the behavioral level we can represent digital system with a single DD or to partition the system into control-flow and data-flow DDs. For illustrative purposes we will use the latter approach here. The control-flow DD carries two types of information: state transition information and path activation information. The state transition information captures the state transitions that are given in the FSM corresponding to the specified system. The path activation information holds conditions associated with state transitions.

Depending on the partition of a system into a network of subsystems, we can represent the whole DD model as a set of DDs, so that for every output of a subsystem a DD will be associated with it.

A test for a system represented by DDs can be created in two parts [15]:

- A *scanning test*, which makes sure that the different functional blocks are working correctly.
- A *conformity test*, which makes sure that the different working modes chosen by control signals are properly carried out.

In [8] it has been shown that in some cases there exists a gap between the fault coverage figures obtained by test sequences generated purely on a high level and those by the gate-level ones. This gap can be reduced by integrating structural information to the test generation process by employing the HTG approach to be discussed here.

The main idea of the HTG technique [10] is to use information from different abstraction levels while generating tests. One of the main principles is to use a modular design style, which allows us to divide a larger problem into several smaller subproblems and to solve them separately. This approach allows the generation of test vectors for the lower level modules based on different techniques suitable for the respective entities.

The HTG algorithm of interest to us generates conformity tests from pure behavioral descriptions. This test set targets errors in branch selection (nonterminal nodes of the DDs). During the second test generation phase the functional blocks (*e.g.*, adders, multipliers and arithmetic and logic units) composing the behavioral model are identified (terminal nodes of the data-flow DDs), and suitable test vectors are generated for the individual blocks. During the block-level test-generation phase each block is considered as an isolated and fully controllable and observable entity; and a gate-level test-generation tool is used for this purpose. The test vectors generated for the basic blocks are then justified and their fault effects propagated in the behavioral model of the circuit under test. In this way we can incorporate accurate structural information into the high-level test pattern generation environment while keeping propagation and justification task still on a high abstraction level. In the following the test pattern generation algorithm is described in detail.

### 5.4.1 Scanning Test

Consider a terminal node $m^{\mathrm{T}} \in M^{\mathrm{T}}$ in $G_{z,k}$, labeled by a functional expression $f_m(Z(m^{\mathrm{T}}))$. To generate a test for the node $m^{\mathrm{T}}$ means to generate a test for the function $f_m(Z(m^{\mathrm{T}}))$.

For generating a test for $f_m(Z(m^{\mathrm{T}}))$ we have to solve two tasks:

1. To activate a path $l(m_0, m^{\mathrm{T}}) \subseteq M$, from the root node $m_0$ of the DD up to $m^{\mathrm{T}}$ by choosing proper values $z(m)^*$ for all the control variables $z(m)$ in the nodes $m \in l(m_0, m^{\mathrm{T}}) \setminus m^{\mathrm{T}}$.
2. To find the proper sets of data values $D = (D^1, D^2, \ldots, D^p)$ for the variables $Z(m^{\mathrm{T}})$ to test the function $f_m(Z(m^{\mathrm{T}}))$.

For executing these two tasks, we can use the following test program:

```
FOR all the values of $t = 1, 2, \ldots, p$
BEGIN
    Load the data registers $Z(m^{\mathrm{T}})$ with $D^t$;
    Carry out the tested working mode at the control values $z(m)^*$ for all $z(m)$,
        $m \in l(m_0, m^{\mathrm{T}}) \setminus m^{\mathrm{T}}$;
    Read the value of $z_k$ and compare it with the reference value $f_m(D^t)$.
END.
```

The task of the scanning test is to detect the faults in registers, buses and data manipulation blocks. In terms of DDs, the terminal nodes are tested by the scanning test.

***Example 5.1:*** We illustrate how a test can be generated for testing the multiplier in Figure 5.2. In the DD of Figure 5.2 we have two terminal nodes with the multiplier function. Let us choose the node $R_1*R_2$ for testing. By activating the path to this node (shown by bold in Figure 5.2) we generate a control word $y_2$, $y_3$, $y_4$ = 0,3,2. To find the proper values of $R_1$ and $R_2$ we need to descend to the lower level (*e.g.*, gate level) and generate test patterns by a low-level ATPG for the implementation of the multiplier. Let us have a test set of $n$ test patterns ($D_{11}$, $D_{21}$; $D_{12}$, $D_{22}$; … $D_{1p}$, $D_{2p}$) generated for the multiplier with inputs $R_1$ and $R_2$.

Based on the above information, the following test program can be used:

```
FOR all the values of $i = 1, 2, …, p$
BEGIN
    Load the data registers $R1 = D_{1i}$, $R2 = D_{2i}$;
    Carry out the tested working mode at the control values $y_2, y_3, y_4 = 0,3,2$;
    Read the value of $R2$ and compare with the reference $D_{1i} * D_{2i}$.
END.
```

### Scanning Test in the Hierarchical Test Generation Environment

One of the most important parameters guiding the design synthesis process is the technology and module library that will be used in the final implementation. By defining the technology and module library, we can have information about the implementation of functional units that will be used in the final design. The HTG algorithm can employ this structural information for generating tests. Tests are generated by cooperation of high-level and low-level test pattern generators. It is usually performed one by one for every arithmetic operator given in the specification.

In the HTG environment we describe here, the algorithm starts by choosing an operator not yet tested from the specification, and uses a gate-level ATPG to generate a test pattern targeting structural faults in the corresponding functional unit. In this approach an ATPG inspired to the Path Oriented Decision Making (PODEM) algorithm is used, but in the general case any gate-level test pattern generation algorithm can be applied. If necessary, pseudorandom patterns can be used for this purpose as well.

The test patterns, which are generated by this approach, can have some undefined bits (don't cares). As justification and propagation are performed at the behavioral level by using symbolic methods, these undefined bits have to be set to a given value. Selecting the exact values is an important procedure, since not all possible values can be propagated through the environment and it can, therefore, lead to the degradation of fault coverage.

A test vector that does not have any undefined bits is thereafter forwarded to a constraint solver, where together with the environmental constraints it forms a test case. Solving such a test case means that the low-level test vector generated can be justified till the primary inputs and the fault effect is observable at the primary outputs. If the constraint solver cannot find an input combination that would satisfy the given constraints, another combination of values for the undefined bits has to be chosen and the constraint solver should be employed again. This process is continued until a solution is found or timeout occurs.

If there is no input combination that satisfies the test case generated, the given low-level test pattern will be abandoned and the gate-level ATPG will be employed again to generate a new low-level test pattern. This task is continued until the low-level ATPG cannot generate any more patterns.

Figure 5.3 illustrates the results of applying the above HTG algorithm to the DIFFEQ benchmark. We have annotated the VHDL behavioral description of the design with the test generation results. With every functional unit (FU), the following information is attached:

1. The total number of stuck-at faults in it, when implemented in the target technology.
2. The number of vectors, which were generated by the gate-level ATPG and successfully justified to the primary inputs and propagated to the primary outputs.
3. The final stuck-at fault coverage for the FU.

As can be seen, fault coverage of FUs differs significantly, depending of the location and type of every individual FU. This information can be successfully exploited at the latter stage of the DfT flow, when selecting modules for DfT modifications.

```
ENTITY diff IS
  PORT
      ( x_in     : IN integer;
        y_in     : IN integer;
        u_in     : IN integer;
        a_in     : IN integer;
        dx_in    : IN integer;
        x_out    : OUT integer;
        y_out    : OUT integer;
        u_out    : OUT integer
     ) ;
END diff ;

ARCHITECTURE behavior OF diff IS
BEGIN
  PROCESS

  variable x_var, y_var, u_var,
     a_var, dx_var  : integer;
  variable t1,t2,t3,t4,t5,
             t6,t7: integer ;
  BEGIN
    x_var := x_in;
    y_var := y_in;
    a_var := a_in;
    dx_var := dx_in;
    u_var := u_in;

    while x_var < a_var loop

      t1 := u_var * dx_var;
        -- Tested 5634 faults
        -- Untestable 0
        -- Aborted 14
        -- Fault coverage: 99.75
        -- Fault efficiency: 99.75
        -- 52 Vectors

      t2 := x_var * 3;
        -- Tested 4911 faults
        -- Untestable 0
        -- Aborted 737
        -- Fault coverage: 86.95
        -- Fault efficiency: 86.95
        -- 11 Vectors

      t3 := y_var * 3;
        -- Tested 4780 faults
        -- Untestable 0
        -- Aborted 868
        -- Fault coverage: 84.63
        -- Fault efficiency: 84.63
        -- 10 Vectors

      t4 := t1 * t2;
        -- Tested 5621 faults
        -- Untestable 0
        -- Aborted 27
        -- Fault coverage: 99.52
        -- Fault efficiency: 99.52
        -- 38 Vectors

      t5 := dx_var * t3;
      -- Tested 5616 faults
      -- Untestable 0
      -- Aborted 32
      -- Fault coverage: 99.43
      -- Fault efficiency: 99.43
      -- 35 Vectors

      t6 := u_var - t4;
      -- Tested 368 faults
      -- Untestable 0
      -- Aborted 60
      -- Fault coverage: 85.98
      -- Fault efficiency: 85.98
      -- 9 Vectors

      u_var := t6 - t5;
      -- Tested 424 faults
      -- Untestable 0
      -- Aborted 4
      -- Fault coverage: 99.06
      -- Fault efficiency: 99.06
      -- 15 Vectors

      t7 := u_var * dx_var;
      -- Tested 1123 faults
      -- Untestable 0
      -- Aborted 4525
      -- Fault coverage: 19.88
      -- Fault efficiency: 19.88
      -- 1 Vectors

      y_var := y_var + t7;
      -- Tested 389 faults
      -- Untestable 0
      -- Aborted 39
      -- Fault coverage: 90.88
      -- Fault efficiency: 90.88
      -- 11 Vectors

      x_var := x_var + dx_var;
      -- Tested 414 faults
      -- Untestable 0
      -- Aborted 14
      -- Fault coverage: 96.72
      -- Fault efficiency: 96.72
      -- 15 Vectors

    end loop ;

    x_out <= x_var;
    y_out <= y_var;
    u_out <= u_var;

    END PROCESS ;

  END behavior;
```

**Figure 5.3.** DIFFEQ benchmark with testability figures for every individual FU

### 5.4.2 Conformity Test

Consider a nonterminal node $m$ labeled by a control variable $z(m)$ in a given DD $G_{z,k}$, representing a digital system with a function $z_k = f_k(Z_k)$. Let $Z = (Z_C, Z_D)$, where $Z_C$ is the vector of control variables and $Z_D$ is the vector of data variables. To generate a test for the node $m$ means to generate a test for the control variable $z(m) \in Z_C$. Suppose that the variable $z(m)$ may have $n = |z(m)|$ different values. For testing $z(m)$ we have to activate and exercise all the proper working modes controlled at least once by each value of $z(m)$. At the same time, for each such working mode, a current state of the system should be generated, so that every possible faulty change of $z(m)$ should produce a faulty next state different compared with the expected next state for the given working mode.

Let us denote by $m^e$ the successor node of the node $m$ for the value $z(m) = e$, where $e = 1, 2, \ldots, n$. For generating a test for $m$ we have to solve the following tasks on the DD:

1. To activate a path $l(m_0, m)\backslash m \subseteq M$ from the root node of the DD up to the node $m$ by choosing proper values $z(m')^*$ for all the control variables $z(m') \in Z_C$ in the nodes $m' \in l(m_0, m)\backslash m$.
2. To activate for all neighbors $m^e$ of $m$ nonoverlapping paths $l(m^e, m^{e,\mathrm{T}})$ from $m^e$ up to the nonoverlapping terminal nodes $m^{e,\mathrm{T}}$ by choosing proper values $z(m')^*$ for all the control variables $z(m') \in Z_C$ in the nodes of $m' \in l(m^e, m^{e,\mathrm{T}})$.
3. To find the proper set of data (the values $z^*$ of the variables $z \in Z_D$), by solving the inequality $z(m^{\mathrm{T},1}) \neq z(m^{\mathrm{T},2}) \neq \ldots \neq z(m^{\mathrm{T},n})$ where $n = |\, v(z(m))|$.

Consider the resulting test as a set of symbolic test patterns $T = \{(z(m) = e, Z_C^*, Z_D^*, z(m^{\mathrm{T},e})) \mid e \in v(z(m))\}$, where $e$ is the symbolic value of the tested variable $z(m)$; $Z_C^*$ is the constant vector of the other control signals corresponding to the set of variables $Z_C \subseteq Z$, and generated by the first two steps of the algorithm; $Z_D^*$ is the constant vector of the data values corresponding to the set of variables $Z_D \subseteq Z$, and generated by the third step of the algorithm; and, finally, $z(m^{\mathrm{T},e})$ is the expected output value of the system corresponding to the value $e$ of the tested control variable $z(m)$. The final conformity test of the control variable $z(m)$ created from the symbolic test pattern $T$ consists of the following program:

```
FOR each value of $e = 1, 2, \ldots, |z(m)|$
BEGIN
   Load the data registers with $Z_D^*$;
   Carry out the tested working mode at the control signals $z(m) = e$, and
      $Z_C^*$;
   Read the value of $z_k$, and compare with the reference value $z(m^{\mathrm{T},e})$.
END.
```

The task of the conformity test is to detect the control faults and the faults in multiplexers. In terms of DDs, the nonterminal nodes are tested by the conformity test.

For example, in order to test nonterminal node IN1 in Figure 5.4, one of the output branches of this node should be activated. Activation of the output branch means activation of a certain set of program statements. In our example, activation of the branch IN1 < 0 will activate the branches in the data-flow DD where $q = 1$ ($A := X$). For observability, the values of the variables calculated in all the other branches of IN1 have to be distinguished from the value of the variables calculated by the activated branch. In our example, node IN1 is tested, in the case of IN1 < 0, if $X \neq Y$. The path from the root node of the control-flow DD to the node IN1 has to be activated to ensure the execution of this particular specification segment and the conditions generated here should be justified to the primary inputs of the module. This process will be repeated for each output branch of the node. In the general case there will be $n(n - 1)$ tests, for every node, where $n$ is the number of output branches.

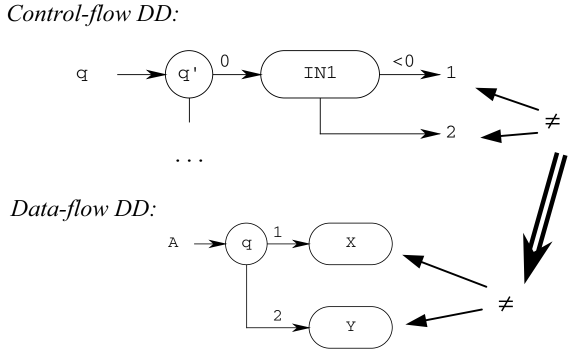


**Figure 5.4.** Conformity test example

***Example 5.2:*** Let us consider how to generate a test program for testing the node $m$ labeled by $y_3$ in Figure 5.2. First, we activate the path $l(m_0, m)\backslash m$, which results in $y_3 = 2$. Then we activate four paths $l(m, m^{e,\mathrm{T}})$ for each value $e = 1, 2, 3, 4$ of $y_3$, which results in $y_1 = 0$ and $y_2 = 0$. Two of the four paths for values $y_3 = 1$ and $y_3 = 2$ are "automatically" activated, since the successors of the node $y_3$ for these values are terminal nodes. The test data $R1 = D_1$, $R2 = D_2$, IN $= D_3$ are found by solving the inequality

$$R1 + R2 \neq IN \neq R1 \neq R1 * R2 \tag{5.1}$$

From the procedure described above, the following conformity test for the control variable $y_3$ is generated:

```
FOR $e = 1, 2, 3$, and 4
BEGIN
    Load the data registers $R1 = D_1$, $R2 = D_2$;
    Carry out the tested working mode at
        $y_3 = e, y_1 = 0, y_2 = 0, y_4 = 2$ and IN = $D_3$;
    Read the value of $R2$, and compare it with the reference value $z(m^{\mathrm{T},e})$.
END.
```

In the case when the control values are data dependent the algorithms then become more complicated, since the data found for nonterminal nodes by activating the paths in the DD should be consistent with data found in processing the terminal nodes.

In the general case, a digital system cannot be represented by a single DD. In this case a system will be represented as a network of components or subsystems where each subsystem is modeled by its own DD. The test sequences generated for a subsystem with its DD by the procedures described above are to be treated as local test sequences. To generate the whole test sequence in a global sense, the classical *fault propagating* and *line justification* tasks should be solved on the system level. For solving these tasks, DDs can also be used.

To justify a value $D$ for a variable $z_k$ represented by a DD $G_{z,k}$, a path should be activated in $G_{z,k}$ from the root node to a terminal node $m^T$ labeled by a register, bus or input variable $z$, and the value $D$ is assigned to $z$. If $z$ corresponds to an input or any other directly controllable point, the line justification task is finished. Otherwise, if $z$ is a register or a bus represented by its own DD $G_z$, the line justification tasks will be iteratively solved for $z$ using the graph $G_z$.

To propagate the fault from the point represented by a variable $z$ through a subsystem which is represented by a DD $G_{z,k}$, a test generation procedure described above should be carried out in $G_{z,k}$ for the node $m$ labeled by $z$. The test generated for the node $m$ is propagating any erroneous value of $z(m)$ to the output variable $z_k$ of the subsystem.

## 5.5 Conclusions

This chapter describes a modeling technique, the DD, which is used to capture a digital design at several levels of abstraction. We illustrate first how DDs can be used to capture a gate-level design, with respect to both functional and structural information. The use of DDs to capture designs at the RT and behavioral levels was then described.

With the help of the DDs, an HTG approach could be developed to generate test patterns efficiently based on information from several abstraction levels. The hierarchical test pattern generation technique described generates test sequences with higher fault coverage than those of a pure behavioral test generator. This improvement in fault coverage has been obtained by integrating structural information coming from lower level design. The algorithm maintains a fast efficacy in terms of execution speed by mainly working at the behavioral level for test vector justification and propagation. In the particular HTG implementation, a constraint-solving algorithm is used to solve the vector justification and propagation problems.

## References

[1] Abraham JA (1986) Fault modeling in VLSI. VLSI testing. North-Holland, 1-27

[2] Brahme D, Abraham JA (1984) Functional testing of microprocessors. IEEE Transactions On Computers, 33(6): 475-485

[3] Bryant RE (1986) Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers, 35(8): 667-690

[4] Ghosh S, Chakraborty TJ (1991) On behavior fault modelling for digital designs. Journal of Electronic Testing: Theory and Applications, (2): 135-151

[5] Giambiasi N, Santucci JF, Courbis AL, Pla V (1991) Test pattern generation for behavioral descriptions in VHDL. In: Proc. of the VHDL conference, 228-234

[6] Gupta AK, Armstrong JR (1985) Functional fault modeling and simulation for VLSI devices. In: 22nd Design Automation Conference, 720-726

[7] Jervan G, Eles P, Peng Z (1999) A hierarchical test generation technique for embedded systems. In: Proc. Electronic Circuits and Systems Conference, 21-24

[8] Jervan G, Peng Z, Goloubeva O, Sonza Reorda M, Violante M (2002) High-level and hierarchical test sequence generation. In: Proc. of IEEE International Workshop on High Level Design Validation and Test, 169-174

[9] Minato S (1996) BDDs and applications for VLSI CAD. Kluwer Academic Publishers

[10] Murray BT, Hayes JP (1998) Hierarchical test generation using precomputed tests for modules. In: Proc. IEEE International Test Conference, 221-229

[11] Raik J, Ubar R (1999) Sequential circuit test generation using decision diagram models. In: Proc. of IEEE Design Automation and Test in Europe, 736-740

[12] Shen L, Su SYH (1988) A functional testing method for microprocessors. IEEE Transactions on Computers, 37(10): 1288-1293

[13] Su SYH, Lin T (1984) Functional testing techniques for digital LSI/VLSI systems. In: 21st ACM/IEEE Design Automation Conference, 517-528

[14] Thatte SM, Abraham JA (1980) Test generation for microprocessors. IEEE Transactions on Computers, 29(6): 429-441

[15] Ubar R (1996) Test synthesis with alternative graphs. IEEE Design&Test of Computers, Spring 1996: 48-57

[16] Ubar R (1998) Multi-valued simulation of digital circuits with structurally synthesized binary decision diagrams. Multiple Valued Logic, 4: 141-157

[17] Ubar R (1998) Combining functional and structural approaches in test generation for digital systems. Microelectronics Reliability, 38(3): 317-329

[18] Ward PC, Armstrong JR (1990) Behavioral fault simulation in VHDL. In: 27th ACM/IEEE Design Automation Conference, 587-593

# 6 Test Program Generation from High-level Microprocessor Descriptions

E. Sánchez, M. Sonza Reorda, G. Squillero

Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy

## 6.1 Abstract

This chapter describes and analyzes a methodology for gathering together test-programs for microprocessor cores during the complete design cycle starting from early design phases. The methodology is based on an almost automatic tool and could be applied to generate test-programs for stand-alone microprocessor cores as well as for these embedded in systems-on-chip. The main idea is to take advantage of all possible microprocessor descriptions delivered through the whole design cycle to generate test-programs able to achieve a high FC% at gate-level. Most of the efforts of the methodology presented are focused on test program generation from high-level microprocessor descriptions. A case study is presented tackling a pipelined microprocessor core.

## 6.2 Introduction

As sketched by the SIA02 roadmap, today most of the integrated circuit (IC) manufacturing cost is brought about by the test processes. Only a few years ago, the testing cost represented a small percentage of the total cost in the whole manufacturing budget, but, among other reasons, the increasing difficulty in generating appropriate testing patterns and the expensive elaboration times required to test an IC raised these costs up to near 70%. Moreover, test methodologies do not progress at the same pace as manufacturing technology does, contributing to enlarging the cost gap. While the production costs continue to go down, the testing cost slope remains flat or trends upward [19].

The problem of testing is especially critical in the case of microprocessors and microcontrollers. Modern designs contain complex architectures taking advantage of the current copiousness of resources brought by technological advances. For example, most of the modern microprocessors are based on pipeline structures; thus, performance and functionality are enhanced, but at the same time the test complexity is increased. Indeed, pipelined and superscalar designs have been demonstrated to be random pattern resistant.

Nowadays, new integration trends make it possible to design an entire system into a single chip, the so-called System-on-a-Chip (SOC). SOCs commonly include one or more microprocessor cores. These cores may be purchased as Intellectual Properties (IPs) from a third-party core vendor, or designed in-house. Microprocessor cores are following the same trend as high-end microprocessors, and quite complex units may be easily found in modern SOCs. The incredible diffusion of these kinds of core is increasing the challenges in the test arena.

Through their lifecycle, microprocessor cores undergo several tests and audits. At earlier stages, validation and verification tests aim at guaranteeing that the final unit conforms to the initial specifications. Each time that a new step into the design cycle is reached, the new design must be verified. Comparing the behavior of the new model with the previous one is a required step, but also the new features should be checked. Once the processor is produced, manufacturing tests must be performed to determine the correctness of the final product. At this point, parametric and functional tests are commonly applied by the manufacturers. Parametric tests guarantee a general acceptance of the Device Under Test (DUT), avoiding, for example, shorts or open circuits. Functional tests check for proper operation of the DUT. The final user may carry out new tests to eventually assert the received unit functionality or to detect possible errors in the microprocessor during its infield lifecycle.

Traditionally, tests have been applied using external Automatic Test Equipments (ATEs) [1]. However, technological progress is pushing up the complexity and operating frequencies of low-end microprocessor cores. It has become apparent that parametric testing alone is not sufficient to achieve the high quality goals required. Moreover, even though ATE effectiveness on applying parametric test is unquestionable, the costs for an ATE able to run at-speed functional tests are becoming prohibitive for manufacturers of moderate quantities of units [19]. As a consequence, the test community headed to alternative solutions.

To overcome these problems, industries are trying to reduce the use of expensive ATEs. One interesting strategy is to perform part of the test plan resorting to the so called Software-Based Self-Test (SBST) [4], where the test consists in a mere set of assembly programs and does not rely on any special test point to force values or observe behaviors during test application. Such programs could be loaded in RAM (*e.g.*, resorting to a DMA or other mechanisms), and executed to test the core. A minimum effort is needed to extract test results.

Clearly, an SBST test is executed at-speed, and requires a very simple ATE and little hardware overhead to download and upload the test information. Furthermore, the SBST is a suitable solution for stand-alone microprocessors as well as for those embedded in an SOC. However, the problem of generating effective test-programs is still open.

This chapter describes a methodology for devising effective test-programs using a test strategy based on an SBST. The resulting methodology is almost automated and takes benefit from the information available in the different design stages, from the Instruction Set Architecture (ISA) to Register Transfer level (RT-level), and eventually netlist. The key idea is to build a set of test-programs through the whole design cycle of the processor, using the microprocessor high-level descriptions delivered during the design process. Indeed, the most of the efforts and analysis in the presented methodology are focused on automatic generation of test-programs for microprocessors at RT-level; in fact, a systematic exploitation of the available coverage metrics at this design abstraction level reports interesting results.

Following the proposed methodology, a complete set of programs suitable to appropriate test the tackled microprocessor core will be available. In addition, the generated test-programs can also be exploited for design verification in the earlier phases of the design.

Additionally, the framework set up for performing the experiments allows a quantitative comparison of the different metrics at RT-level. A detailed analysis of the effectiveness of programs devised for maximizing different metrics, and their usability for testing is also reported.

The chapter organization is as follows: the next section presents an outline about the state of the art in test-program generation for microprocessors; Section 6.4 describes the proposed methodology in detail. Once the methodology basic elements have been described, an experimental setup developed to analyze the method suitability is presented in Section 6.5: a pipelined microprocessor core is used as a case study and an evolutionary tool based on the genetic programming paradigm is used as an automatic test-program generator. The experimental results are shown in Section 6.6. Finally, Section 6.7 presents an analysis of the results obtained and concludes the chapter.

## 6.3 Microprocessor Test-program Generation

Microprocessor test-programs have traditionally been devised resorting to functional approaches based on exciting functions and resources. The canonical methodology is described in [22]; however, it involves a high amount of manual work performed by skilled programmers, and does not provide any quantitative measure about the gate-level fault coverage obtained. In addition, this methodology leads to very long test sequences which, for example, test all registers, test all transfer paths between registers, test every instruction with every possible operand, and so on.

Different approaches (such as [3]) proposed interesting techniques for efficient compilation of self-test programs, but they left the responsibility for generating the self-test programs to the test engineers.

In [21] the program VERTIS was presented, which is able to generate both test and verification programs based only on the processor's ISA. The final program exploits almost all the possible operands for each instruction of the GL85 microprocessor, leading to very large programs. The VERTIS program was compared against the patterns generated automatically by two Automatic Test Pattern Generation (ATPG) tools, giving excellent results. However, users need to determine the heuristics to assign values to instruction operands to achieve high stuck-at fault coverage. In some cases, this might not be a trivial task.

Chen and Dey [5] propose DEFUSE, a deterministic method to generate test-programs able to reach good fault coverage on the Arithmetic and Logic Unit (ALU) of a microprocessor, and to compact the result. The approach is very effective with combinationally testable parts (*i.e.*, ALUs), but shows some limitation when hard-to-test sequential modules (*e.g.*, control units) are addressed. On the other hand, [2] is based on generating random sequences of instructions. It is able to attain a fairly high level of fault coverage; however, it assumes that all instructions are single-cycle and buses are never floating. Both approaches require the insertion of Built-In Self-Test (BIST) circuitry.

Regarding test application, a possible architecture supporting software-based self-test solutions was described in [14] and [9]. To make this approach feasible, RAM of sufficient size should be available on the SOC and easily accessible externally. An ATE can load into the memory the test-program when required, and the processor core can execute it. Test execution is always performed at-speed, independently of the speed of the mechanisms used for loading the RAM and checking results.

Corno *et al.* [9] propose a semi-automated approach to test-program generation based on a library of macros. A genetic algorithm chooses those macros. The approach is shown able to attain reasonable fault coverage (85%) on a common microprocessor core, and requires no additional hardware or scan structures. However, test generation relies on a library carefully compiled by experts.

Kranitis *et al.* [11] describe a methodology that allows devising an effective test-program for a microprocessor core. However, the method requires that test engineers create deterministic test patterns to excite the entire set of operations performed by each component of the core.

Paschalis *et al.* [16] propose an evolution of [11]: a component-based divide-and-conquer approach is proposed for on-line test generation for microprocessors. Program generation is based only on the processor ISA and RT-level description. The methodology is developed going through three phases: information extraction, classification of the processor components, and test code development for components. At the end, the authors take advantage of the deterministic routines developed earlier to tackle specific components.

An automated functional self-test method based on generation of random instruction sequences with pseudorandom data, generated by software Linear Feedback Shift Register while the approach uses the on-chip cache to apply these, is presented in [15]. The results obtained reflect the method effectiveness; however, strong constraints, such as that the loaded program can produce neither cache misses nor bus cycles, make the method hardly transportable.

More recently, in [18] a mixed methodology using SBST and low-overhead BIST has been applied to test embedded Digital Signal Processors (DSPs) core components. Test routines are generated taking into account the instructions' observability and controllability over the component. These test routines are executed within a loop with different random numbers delivered by the appropriate hardware-inserted random generators to enhance the final fault coverage. As the results show, the method is suitable for functional components but not for control components.

An almost automatic framework was proposed in [7]. The approach is based on an evolutionary algorithm and it is capable evolving small test-programs able to capture quite interesting target corner cases. The approach demonstrated its effectiveness when it was compared against an instruction randomizer method tackling a pipeline microprocessor. At the end, not only sharper programs were developed, but also smaller ones.

Available high-level routines have also been used, but despite the effortlessness, this is not a good solution for test. Owing to the intrinsic nature of the algorithms and of compiler strategies, these programs are seldom able to excite all functionalities and do not take into account observability.

On the other side, generating verification programs for microprocessor cores could be performed using two different approaches: formal and simulation based.

Formal methods try to verify the correctness of a system by using mathematical proofs, whereas simulation-based design verification tries to uncover design errors by detecting a circuit faulty behavior when deterministic or pseudorandom tests are applied. Formal methods implicitly consider all possible behaviors of the models representing the system and its specification, so the accuracy and completeness of the system and specification models, as well as required computation resources, are a fundamental limitation. On the contrary, simulation-based methods do not suffer from the same constraints, but they can only consider a limited range of behaviors and will never achieve 100% confidence of correctness.

Formal verification methods for complex microprocessor designs have been targeted by several authors, for instance [10], [24] and [25]. Although results were significant, these methods require considerable human efforts.

Simulation-based methods have demonstrated their suitability for verifying complex microprocessors; [20] proposed a technique where the processor itself generates a test at run-time by self-modifying code. Similarly, [23] showed a method for generating instruction sequences for validating the branch prediction mechanism of the PowerPC604. Generated sequences are very effective, but methodologies exploit a deep knowledge of the target processors and cannot be easily applied on general designs.

In [8], the experimental results show that automatic simulation-based methods are able to reach high coverage levels of the RT-level processor description, despite the fact that in that case the microprocessor tackled was quite complex.

## 6.4 Methodology Description

The proposed methodology is based on a step-by-step approach that allows generating and collecting test-programs through the whole design cycle. Test-program generation is performed each time a new microprocessor model is delivered; thus, it is not necessary to wait to the end of the design cycle to start the generation of programs. Moreover, being a cumulative process that tackles high-level descriptions, the approach minimizes the use of explicit low-level fault tions, the approach minimizes the use of explicit low-level fault simulation to drive the test-program generation, greatly reducing computational efforts. In fact, fault simulation is only used at the end of the generation to complete the set of test-programs. Figure 6.1 is a graphical representation of the possible progress of the final fault coverage (FC) percentage (FC%) at gate-level obtained by the test-programs devised in the different phases.

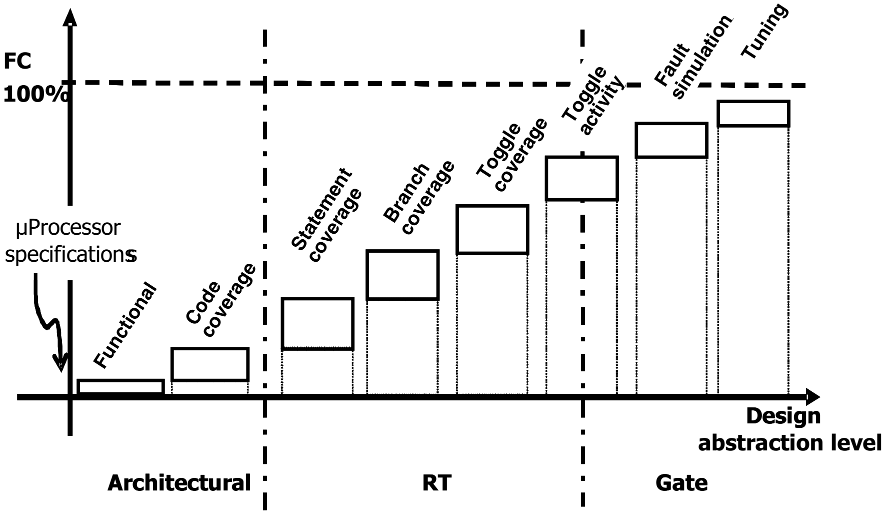


**Figure 6.1**. Qualitative description of the methodology

To determine if a logic circuit is good, patterns or test vectors are applied to the circuit inputs, and then the circuit responses are compared with the expected outputs. If the responses match, the circuit can be considered as good. However, the results of the test process are directly correlated with the quality of the patterns [1]. The performance measure of the test patterns is commonly called the FC. This percentage value represents the patterns' ability to detect specific faults in the logic circuit. A fault is a model of one or more possible physical defects. In most cases, when an FC value is provided, the single stuck-at fault is chosen as the fault model. However, other fault models can be adapted, such as the delay fault model.

A general microprocessor design lifecycle is shown in Figure 6.1. The whole cycle has been split into three successive macro phases. Each phase is characterized by a different kind of design abstraction model: architectural, RT level and gate level.

A pure simulatable model, like the one embedded in an Instruction Set Simulator (ISS), characterizes the first phase; architectural simulatable models, such as high-level programs possibly catching some peculiar aspects of the internal design, may be found too. In the second phase, more structured models such as RT-level descriptions are available; hybrid models, where some blocks have been detailed down to gate level, but others are still at the RT level, could be included in this phase. Behavioral simulators drive this section of the process. Finally, in the last phase of the design, before the manufacturing stage, the processor netlist is assumed available.

Figure 6.1 gives a qualitative idea of the FC% that could be obtained by generating test-programs using different microprocessor models; these percentages reflect the potential performance of test-programs generated on each stage. As estimated, test-programs generated in the early stages provide poorer results in terms of FC%; however, these collected programs are a good initial point regarding the entire process. In particular, this chapter shows that test-programs generated in the second part of the design cycle, when RT level and hybrid descriptions are available, are able to reach an important contribution of fault coverage regarding the complete set of programs.

The descriptions available in the final phase lead to a more expensive test-programs generation process; consequently, this phase should mainly be devoted to completion and tuning regarding FC%.

As outlined before, the main goal of the generation process is to gather a set of programs able to reach a high FC%; in the following, a more detailed explanation about both models and metrics used in each stage is presented.

### 6.4.1 Architectural Models

In the first steps of a microprocessor design, a model may not be available, and designers can only rely on the ISS based on the microprocessor ISA. Later, a more complete simulatable model may be built in some high-level language. This situation is depicted in Figure 6.2.

Simulatable models are used directly to verify the system behavior against the specifications. Compatibility proofs may be performed to check, for example, whether new microprocessor enhancement conforms with previous version specifications. Most of the programs used to audit those models are functional ones.

Usually an ISS is available from the first design stages. The ISS is able to simulate the program flow execution on the microprocessor and occasionally could even be considered a kind of specification. At this time, the ISA could be evaluated through functional programs. In the first stage, verification programs are generally hand written; those programs exercise specific functionalities according to the design engineer experience; for example, very simple programs containing all possible instructions or performing an intricate mathematical function can be developed.

Since functional or hand-written programs do not target the FC, by examining those programs' results against the FC% it is possible to observe very low coverage; however, most of them cover special corner cases.

More interesting, simulatable models are sometimes available, especially when microprocessor complexity is higher. Those models are typically processor high-level descriptions composed of a set of functional boxes that more closely match the planned architecture. Normally, these blocks represent complete functional entities like closed and independent blocks, but different solutions are also possible at this description level.

Devising test-programs at this level is not a trivial task. Currently, test-programs for such simulatable models can target code coverage metrics such as line coverage of the program implementing the microprocessor, sometimes called architectural coverage.

There is not a very strong correlation between simulatable models' metrics and final processor faults, mainly because the observability is lacking. Thus, the FC% results obtained using these programs are low.

Accordingly, in the effort to increment test-program effectiveness, it is possible to use clever metrics such as the Toggle Activity (TA) undergone by all interconnection paths between functional blocks. However, using new metrics could require one to instrument the architectural model with new macro blocks able to compute the specific metric, which do not belong to the final processor description, and in addition could decrease the model performance. Another option to enhance test-program generation at this stage consists in the use of more detailed microprocessor descriptions, but usually they are not available.

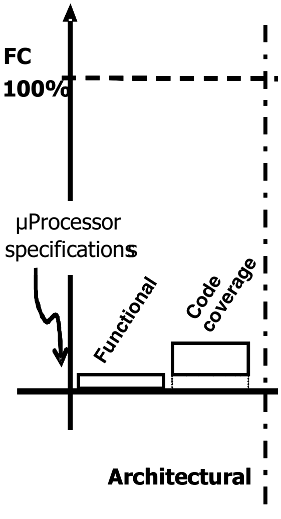


**Figure 6.2**. Architectural models stage

### 6.4.2 Register-transfer-level Models

Most of the efforts in the methodology presented are on test-program generation for RT-level models, because those microprocessor representations are light enough to work on today's simulation tools. Figure 6.3 is a representation of this stage.

Different logic models of the processor, such as RT-level descriptions or hybrid RT-gate level ones, are found in the middle of the design cycle. As a part of their model verification process, and before manufacturing, a set of verification programs is generated to discover design flaws. Usually, those programs aim at exciting almost all the microprocessor functionalities.

### *Pure Register-transfer-level Models*

RT models are more detailed microprocessor descriptions than simulatable ones. Each time the design process advances, new RT-level models are delivered corresponding to more and more detailed structures belonging to the microprocessor architecture. To assert whether those models correspond to the specifications, early audits are performed using verification programs.

Generation of verification programs is a time-demanding task; therefore, those programs represent valuable material for the whole design cycle. According to this methodology, if verification programs have been generated using a refinement method, these programs could be suitable test-programs.

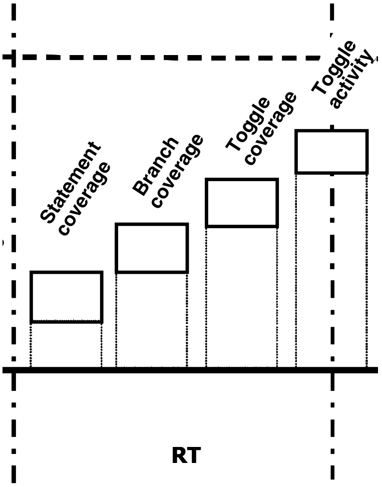


**Figure 6.3.** RT model stage

The core provider could supply verification programs, but in the case they are not available, automatic tools such as those described in [21] or in [7] can be used to perform the task. Independently from the origin, formal and software-based methods provide sets of programs able to cover special corner cases of the processor.

Targeting verification, statement coverage (SC) has until now been the most popular coverage metric to evaluate the effectiveness of programs. However, many authors hold that it is not possible to accept a single coverage metric as the most reliable and complete one [6]; thus, a 100% coverage on any particular metric cannot guarantee a 100% flaw-free design. Nowadays, thanks to modern logic simulators' features, different metrics can be exploited to guarantee better performance of the test-programs devised at logic stages. Therefore, the verification trend is to combine multiple coverage metrics together to obtain better results. However, not all metrics could be sensibly exploited in the earlier stages of the design flows. Consequently, it is extremely useful to complete a verification set to reach complete coverage on different metrics.

Currently RT-level simulators support an ample set of coverage metrics useful to assert that the test patterns exercise the circuit design thoroughly. Since these simulators are not devised specifically as debugging instruments, designers are pushed to extract code coverage statistics to ensure verification process. The coverage metrics commonly available in commercial simulators are:

- **Statement coverage** is the most basic form of code coverage: SC is a measure of the number of executable statements within the model that have been exercised during the simulation run. Executable statements are those that have a definite action during runtime and do not include comments, compile directives or declarations. SC counts the execution of each statement on a line individually, even if there are multiple statements on that line.
- **Branch coverage** reports whether Boolean expressions tested in control structures (such as the if-statement and while-statement) evaluated to both true and false. The entire Boolean expression is considered a true-or-false predicate regardless of whether it contains logical-and or logical-or operators. Branch coverage is sometimes called decision coverage.
- **Condition coverage** can be considered as an extension of branch coverage: it reports the true or false outcome of each Boolean sub-expression, separated by logical-and and logical-or if they occur. Condition coverage measures the sub-expressions independently of each other.
- **Expression coverage** is the same as condition coverage, but instead of covering branch decisions it covers concurrent signal assignments. It builds a focused truth table based on the inputs to a signal assignment using the same technique as condition coverage.
- **Toggle coverage** reports the number of bits that toggle at least once from 0 to 1 and at least once from 1 to 0 during the execution of a program. At the RT-level, registers are targeted and, since RT-level registers correspond to memory elements with an acceptable degree of approximation, the toggle coverage is an objective measure of the activity of the design. Indeed, this is a very peculiar metric and can be sensibly used in all late stages of the design cycle.

The metrics described stress different aspects of the RT-level microprocessor description. On the one hand, SC, branch coverage and condition coverage deal with the test-program's ability to cover most of the code lines, focusing on program flow. On the other hand, expression coverage and toggle coverage monitor the information behavior through the program execution; they thus check both the microprocessor variables and signals.

In our approach we suggest taking advantage of the available coverage metrics in modern simulators and firstly face the metrics regarding the program flow, and in a refinement phase the use of metrics observing internal information.

Of course, this must be a gradual process that starts maximizing the SC of the microprocessor description; then, when a satisfactory program or set of programs has been obtained, the next metric to deal with is the branch coverage. But this new program generation must not start from scratch; instead, a kind of coverage grading could be devised. The performance metric of the new program set will not be the total branch coverage, but the branch coverage without the branches already covered by the first program set.

In the same way, a third program or set of programs could be developed targeting the condition coverage metric.

The second phase consists of the generation of suitable programs able to attain high coverage regarding data elements inside the microprocessor description. To accomplish this goal, expression coverage and toggle coverage will be used as performance metrics. Again, the program generation can start from the coverage level reached by the previous programs.

### Hybrid Register-transfer–Gate-level Models

When coming closer to the final design stage, the microprocessor descriptions could contain mixed modules; some of them, for example, can be gate-level representations of specific blocks, but others could remain at RT-level. Thus, the boundary between RT-level and gate-level descriptions is not quite sharp.

When mixed microprocessor descriptions are offered, an additional stage of test-program generation can be devised. In this case, the TA related to all the signals belonging to the components at the gate-level becomes a suitable performance metric for this microprocessor description.

Again, a new set of programs can be generated starting from the results obtained by the previously stages. Consequently, starting from a pre-obtained coverage level decreases the time demand to generate complementary programs.

Once the step-by-step sequence described is devised at this abstraction design level, a set of suitable programs for verification and testing has been acquired. In this way, considerable quantities of development time have been saved because generated programs can be used twice. However, the test-program generation process is not still complete.

### Gate Models

The following microprocessor description is usually represented by a complete netlist interconnecting all the gates of the whole system. This description is the real gate-level or transistor level, also called the component-level description. The different types of fault available in these models are known as technology-dependent faults.

The higher is the FC% obtained by the test-programs, the higher is the reliability of the design experiments. Then, before manufacturing, a suitable set of test-programs must be available.

As described in the previous sections, a set of test-programs had been collected through the processor design cycle; however, it is possible that the maximum FC has not been reached by the set of programs gathered. Then a completion and tuning processes must be performed at this level.

The first step is to perform a fault grading using the set of accumulated programs. Thus, an updated scenario about the remaining faults is presented. At this point, a new test-program generation process can be implemented using as feedback value the FC% regarding the uncovered faults. This process could demand an excessive amount of time, and so it might not be possible to perform this program generation loop from scratch.

The resulting programs of this process are also collected test-programs; their performance depends on the available time to carry out the generation; usually, heavy time restrictions pushed by the time to market guide this generation.

At the last stage, an expert engineer performs the final tuning phase. Then, hand-written programs will tackle specific uncovered faults. This practice is again very expensive and very time demanding because it relies on a skilled engineer.

## 6.5 Case Study

As a case study, the proposed methodology was implemented by tackling a pipelined processor. The core is available in two different descriptions: by the first is an RT-level microprocessor description and the second is the microprocessor netlist. An automatic test-program generator tool called µGP was exploited, following the step-by-step method.

In the following, when an FC% value is provided, the single stuck-at fault is chosen as the fault model.

### 6.5.1 Processor Description

The processor chosen is called PLASMA. This is a free microprocessor core available in [17]. The microprocessor conforms to the MIPS I™ architecture [13], and supports interrupts and all MIPS® user-mode instructions except unaligned load and store operations. The original core is implemented in VHDL with a three-stage pipeline.

The PLASMA core presents a 32-bit architecture, 32 general-purpose registers forming the register bank, and additionally two specific registers for the multiplication and division operations; the RAM memory available is addressed by a 13 bits address bus and the Coprocessor 0 (CP0) is incorporated on the CPU chip and supports the virtual memory system and exception handling. CP0 is also referred to as the System Control Coprocessor.

A block diagram description of the PLASMA core is presented in Figure 6.4.

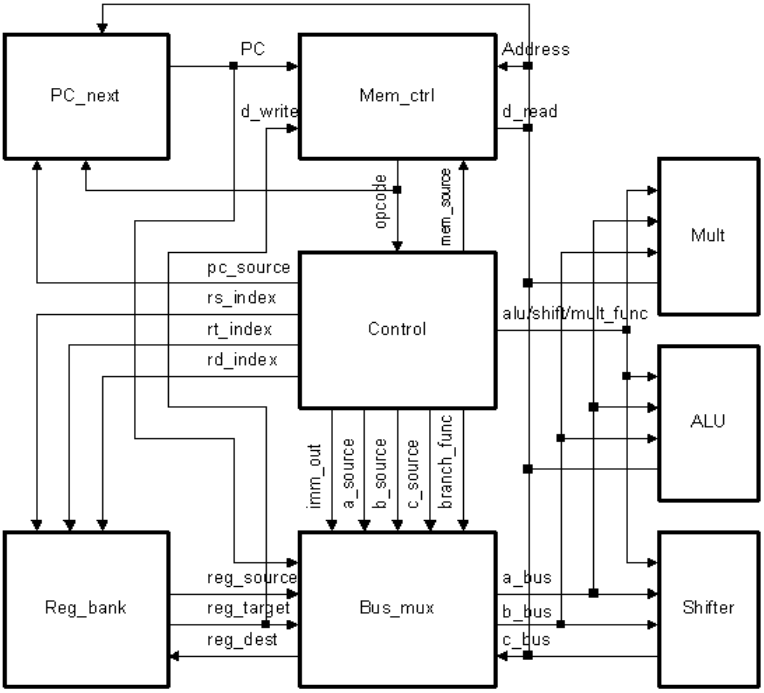


**Figure 6.4.** PLASMA block diagram

Regarding the ISA, the microprocessor supports the following kinds of instructions:

- 14 load and store
- 39 arithmetic-logic
- 23 jump and branch
- 2 coprocessor instructions.

No floating-point instructions are supported by this microprocessor.

### *PLASMA Descriptions*

The original PLASMA core is an RT-level microprocessor description available in VHDL. Table 6.1 is a general description of the core at this level.

Once synthesized, the PLASMA gate-level implementation consists of about 37K gates (1466 are flip-flops). The complete fault list consists of 98,140 permanent single-bit stuck-at faults (95,810 of them are detectable). Table 6.2 summarizes the characteristics of the gate-level description.

**Table 6.1.** PLASMA: RT-level description characteristics

| PLASMA RT-level description | |
|---|---:|
| Active statements | 720 |
| Active branches | 379 |
| Active conditions | 40 |
| Active expressions | 43 |
| Nodes at RT-level | 1436 |

**Table 6.2.** PLASMA: gate-level description characteristics

| PLASMA gate-level description | |
|---|---:|
| Gates | 36,991 |
| Flip-flops | 1,466 |
| Nodes | 17,290 |
| Faults | 98,140 |

### 6.5.2 Automatic Tool Description

This section describes µGP. This is an evolutionary tool based on the genetic programming paradigm.

µGP is able to generate Turing-complete assembly programs. These are generated for a specific target microprocessor, and take advantage of the assembly syntax, exploiting addressing modes and instruction set asymmetries.

µGP has mainly been used for test-program generation, but it is flexible enough to allow many other activities to be tackled. Indeed, it is not even limited to assembly program generation, but it can be used to generate any kind of source code with the same type of syntactic limitations (*e.g.*, state machine descriptions, trees, and so on). The general architecture of µGP is shown in Figure 6.5.

This flexibility comes from the subdivision of µGP into three clearly separated blocks: an evolutionary core, an instruction library, and an external evaluator. The evolutionary core cultivates a population of individuals. It uses self-adaptation mechanisms, dynamic operator probabilities, and dynamic operator strength. The instruction library is used to map individuals to valid assembly language programs. It contains a highly concise description of the assembly syntax or more complex, parametric fragments of code. Finally, the external evaluator simulates the assembly program, providing the necessary feedback to the evolutionary core.

Individuals (representing programs) are stored as loosely linked Directed Acyclic Graphs (DAGs). Each DAG can be considered a collection of sub-DAGs, each one belonging to a defined frame. Every frame, in turn, directly maps into a different program section.

Every frame also contains a prologue and an epilogue, to cater for the declarative parts of the different sections and also for general call/return standard procedures.

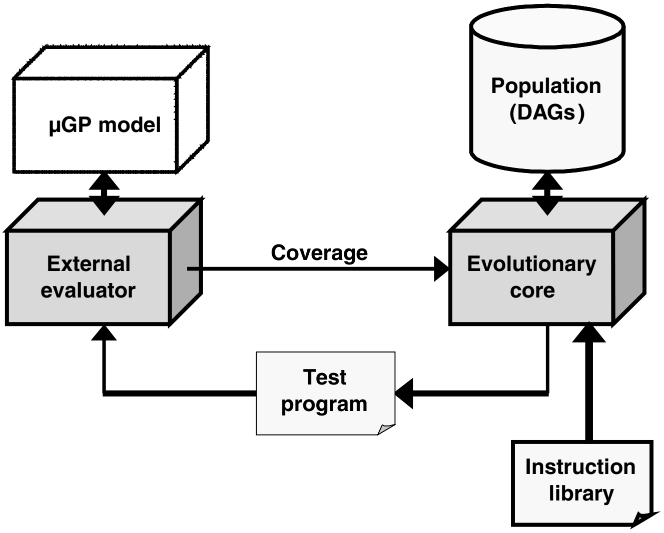


**Figure 6.5.** µGP general architecture

### *Instruction Library*

The instruction library describes the assembly syntax, listing each possible instruction with the syntactically correct operands. It is written as a collection of macros. All these macros, including prologue and epilogue, can correspond to a series of zero or more assembly instructions.

The instruction library also contains the specification of all frames and the list of possible instructions in each frame.

This allows the use of subroutines, data segments and interrupt handlers in the generated programs, fully exploiting the target microprocessor's functions.

A different set of instructions can be specified in every program section, thus allowing (or avoiding) certain program constructs in specific contexts, *e.g.*, allowing the use of supervisor-level instructions in an interrupt handler only.

### *µGP core*

The evolutionary core is able to generate new individuals from an existing population by means of mutation and crossover operators. It implements a ($\mu + \lambda$) evolution strategy. This means that a population of $\mu$ individuals is cultivated, and $\lambda$ genetic operators are applied over the population to obtain an offspring.

The µGP core can generate a random starting population, load an existing one, or a combination of both.

Several different genetic operators are defined in the µGP core, belonging to two main groups: crossover operators and mutation operators. A mutation operator can generate a new individual starting from just one parent, while crossover operators need two. The genetic operators to use are randomly selected according to their activation probabilities.

The parents needed for every operator are chosen using tournament selection, with tournament size $\tau$: $\tau$ individuals are randomly selected and the best one is chosen.

The activation probabilities are self-adapted, as is the size of the tournament.

After creation the new individuals are added to the population and evaluated, then the best $\mu$ individuals of this population are kept for the next generation.

This process goes on until a predefined number of generations is reached, a specified fitness value is obtained, or a steady state is found, which means that evolution is performed for a predetermined number of generations without any improvement in the best fitness value.

Currently, two crossover operators and four mutation operators are implemented, but more may be added in the future.

A number of techniques have gradually been added to the µGP core to improve its performance: clone detection (and optional extermination) to avoid evaluating identical individuals and to preserve diversity in the population; a fitness hole in the tournament selection, again to preserve genetic variability; parallel evaluation; a bigger initial population to exploit the random phase better.

#### *External Evaluator*

The external evaluator is in charge of computing the fitness of the individuals generated. It is external to keep the µGP approach flexible and generic. It is invoked by a system call, and usually has the form of a small script or program that in turn launches a simulator, emulator or the like, collecting and elaborating its results.

In the specialized fields of microprocessor test and validation, this usually means starting a fault simulator or a model simulation to extract different coverage metrics, such as SC or TA, but the method is flexible enough to allow other very different activities to be performed.

### 6.5.3 Experimental Setup

Exploiting both the microprocessor descriptions and the automatic test-program generator, a sequence of experiments as launched obeying the proposed rules of the step-by-step method.

The elements used to devise the experiments are as follows:

- The µGP (*i.e.*, the test-program generator): it consists about 10,000 lines of ANSI C. No special modifications were required to perform this set of experiments.
- External evaluators: two computer-aided design tools were required to complete the µGP generation loop:
  - Modelsim v5.8b by Mentor Graphics at RT-level.

- Faultsim 1999.10-TG4.1-2150 by Synopsys for gate-level fault simulations.

- The microprocessor core: two PLASMA descriptions, mentioned previously.
- The instruction library: including about 60 macros and devised in two programmer working days.

In addition, to interface the μGP code with the external evaluators, a few scripts accessing the different metrics were required.

All experiments were run on a Sun Enterprise 250 with two 400 MHz UltraSPARC-II CPUs and 2 Gb of RAM.

For the completeness of the method, test-programs were devised using both manual and automatic methods considering the actual microprocessor design process. At the end of the process, when gate-level stuck-at FC% was targeted, the observability points used were the buses around the register bank.

## 6.6 Experimental Results

Table 6.3 shows the results obtained through the complete test-program generation process. It is necessary to highlight again that the process was performed as a cumulative collection of test-programs.

Table 6.3 reports the results of the complete step-by-step test-program collection process. The first column, called *Generation process* describes the method used for program generation of each step. *Manual* means that engineers have generated those programs, while *μGP* denotes automatically generated programs.

The second column illustrates the level of abstraction of the microprocessor description, ranging from the ISA to gate-level. Taking advantage of our automatic test-program generator, *i.e.*, the μGP, in our collecting approach, automatic test-program generation can be applied at all description levels; however, manual generation is only allowed at the beginning and at the end of the process. In the specific case study, the automatic generation of test-programs starts at the RT-level of microprocessor description.

The *Metric* column shows the target metrics used to generate test-programs. These metrics were described previously as those suitable for each microprocessor design stage. Since no PLASMA architectural models are available, it was not possible to face this phase.

Along with the information about the final FC obtained in the cumulative process, for each set of test-programs the total programs size, expressed as number of instructions, and the time to develop the appropriate set, in days, are presented in the columns 4 and 5.

The final resulting test-set is composed of 15 test-programs: the first one, at ISA level, was developed targeting quite simple functionalities. At RT-level, eight programs compose the entire set, and all of them were created exploiting the automatic tool; out of the eight, three programs were devised targeting the SC, one the branch coverage, one the condition coverage, two the expression coverage, and the last one aimed at maximizing the toggle coverage. At gate-level, three programs exploit the μGP potentiality; among these, two maximize the TA, but one faces directly the fault coverage. A skilled engineer finally constructed the three remaining programs.

The whole test-set elaboration process requires about 80 working days; however, as sketched before, test-programs generated up to the logic design macro stage were also exploited in the verification process.

**Table 6.3.** Test-program generation results

| Generation process | Processor description | Metric | Size [# inst] | Time [days] | FC [%] |
|---|---|---|---|---|---|
| Manual | ISA | Functional | 57 | 7.0 | 27.12 |
| µGP | RT level | SC | 1325 | 5.0 | 41.92 |
| µGP | RT level | Branch Coverage | 344 | 2.1 | 44.31 |
| µGP | RT level | Condition Coverage | 1187 | 2.4 | 44.81 |
| µGP | RT level | Expression Coverage | 2017 | 2.7 | 46.18 |
| µGP | RT level | Toggle Coverage | 226 | 2.9 | 46.28 |
| µGP | Gate level | TA | 514 | 8.1 | 70.36 |
| µGP | Gate level | Fault simulation | 260 | 21 | 83.57 |
| Manual | Gate level | Tuning FC | 720 | 28 | 91.97 |

Trying to understand where test-programs better stress the unit under evaluation, the following tables present the microprocessor split into its component blocks, showing the FC% of each block achieved by the test-programs.

Table 6.4 shows the performance of the test-programs generated at high description levels starting from the functional one, up to the program automatically generated to maximize the toggle coverage. Additionally, the percentage of faults contained in each unit is supplied in the first column.

As mentioned earlier, the functional program performance is poor; from Table 6.3 it is possible to see that this program never achieves more than the 50% of FC in any microprocessor module. These kinds of program are never devised as test programs, but occasionally they can get together interesting pieces of code that are able to exercise microprocessor singularities, such as the corner cases in the ALU behavior known only by the implementer engineer. In this way, these programs are still suitable for inclusion in the complete test-set of programs.

The PLASMA microprocessor components could be divided into control and functional units; additionally, the core description allows the pipeline to be observed as a singular module. The control units and the pipeline are usually devised as sequential modules conformed by several states and without large combinational parts; these modules are able to handle the complete microprocessor design on each particular situation. Usually, these units present a pattern resistance behavior when they are tested. Thus, it is interesting to see that satisfactory performance was achieved in the early stages dealing with these components. In fact, the most important blocks, such as the memory controller, the control and the pipeline, overcome 70% of FC. On the contrary, the program counter logic and the bus multiplexer have not achieved high coverage because, on the one hand, the final programs do not exercise all possible memory locations nor execute all possible jumps. On the other hand, not all the possible operand combinations were used.

**Table 6.4.** High-level metrics vs gate-level stuck-at FC%

| Module | faults [%] | Funct [%] | µGP-ST [%] | µGP-BR [%] | µGP-CO [%] | µGP-EX [%] | µGP-TX [%] |
|---|---|---|---|---|---|---|---|
| pc_next | 2.33 | 44.49 | 50.04 | 50.04 | 50.09 | 50.09 | 50.09 |
| mem_ctrl | 5.68 | 43.02 | 62.29 | 66.83 | 66.94 | 70.03 | 70.12 |
| control | 1.51 | 49.65 | 81.54 | 84.02 | 85.20 | 85.55 | 85.55 |
| reg_bank | 49.36 | 31.18 | 40.66 | 44.59 | 45.40 | 47.32 | 47.47 |
| bus_mux | 3.64 | 42.98 | 59.89 | 60.72 | 61.07 | 62.70 | 62.82 |
| alu | 6.36 | 47.29 | 65.15 | 66.20 | 66.34 | 67.02 | 67.20 |
| shifter | 5.20 | 18.70 | 38.09 | 38.09 | 38.71 | 38.71 | 38.71 |
| mult | 22.31 | 1.14 | 23.02 | 23.02 | 23.03 | 23.03 | 23.03 |
| pipeline | 3.62 | 46.40 | 68.51 | 69.84 | 70.59 | 74.48 | 74.48 |

The results obtained in dealing with the control and pipeline modules denote that the test-programs generated at the RT-level facing a defined set of coverage metrics are able to stimulate most of the gates included in the synthesized modules, because those modules are complex sequential structures that require appropriate instruction successions to be excited and those special pieces of code can be obtained when high-level coverage metrics are faced.

Dealing with functional blocks, such as the register bank and the multiplier, the program's performance is not exceptional, as is shown in Table 6.4. Generally, these modules are homogeneous combinational circuits, whose RT-level descriptions do not include much detail about the final gate-level implementation; thus, coverage metrics used at RT-level are not able to provide enough information to generate satisfactory test-programs. Actually, as mentioned in [12], such homogeneous circuits can be well excited using deterministic subroutines.

Table 6.5 focuses on test-programs generated at gate-level using both automatic and manual techniques. Table 6.5 also reports the microprocessor block details.

In the Table 6.5, the third column was added to ease the comparative analysis between results obtained at RT and gate levels.

Table 6.5 shows an interesting performance improvement in the FC reached by test-programs generated at gate-level. However, as mentioned before, along with this progress, test-programs generation at low levels requires time-expensive processes. In fact, the elaboration time to obtain the *µGP-TA* programs is at least three times longer than for those generated at RT-level.

**Table 6.5.** Low-level metrics vs gate-level stuck-at FC%

| Modules | Faults [%] | µGP-TX [%] | µGP-TA [%] | µGP-FC [%] | TUNING [%] |
|---|---|---|---|---|---|
| pc_next | 2.33 | 50.09 | 72.31 | 75.13 | 59.05 |
| mem_ctrl | 5.68 | 70.12 | 74.64 | 86.93 | 77.10 |
| control | 1.51 | 85.55 | 89.83 | 92.39 | 81.33 |
| reg_bank | 49.36 | 47.47 | 77.21 | 90.92 | 97.32 |
| bus_mux | 3.64 | 62.82 | 69.90 | 76.38 | 74.40 |
| alu | 6.36 | 67.20 | 89.40 | 93.26 | 96.98 |
| shifter | 5.20 | 38.71 | 44.36 | 69.59 | 65.13 |
| mult | 22.31 | 23.03 | 44.50 | 66.91 | 80.68 |
| pipeline | 3.62 | 74.48 | 89.48 | 92.04 | 91.70 |

Comparing the columns *µGP-TX* and *µGP-TA* in Table 6.5, it is possible to observe that in all control-oriented modules, like *pipeline* and *control unit*, the increase in FC% is limited to about 10%. On the other hand, in functional blocks involving relevant combinational blocks, like *alu*, the increase is much higher. In such blocks, the gate-level TA is able to provide better information about internal structures, allowing the automatic test-program generator to explore the real circuit more deeply. A notable exception is the *program counter logic*, where the increase can be more easily explained with the increased length of test programs, since only long programs are able to activate almost all bits in the program counter.

When comparing *µGP-TA* with *µGP-FC* in Table 6.5, a general FC% improvement is reached in all modules. This time, the selected metric is directly the FC obtained by the test-program. The fundamental idea in this stage is the inclusion of the observability as part of the test-program generation loop. As mentioned previously, the buses around the register bank were selected as observability points for the test evaluation. Again, the best improvements are observed in the register bank and the shifter. However, control and pipeline blocks achieve better performance too.

At the end of the automatic test-program generation process, the control and pipeline modules are still the best covered by the generated programs. The tuning process then performed by an expert engineer looks to improve the performance in three specific functional modules: the register bank, the ALU and the multiplier. Those modules were selected as the tuning target because all of them conform more than 70% of all microprocessor faults. In this fashion, three new programs are devised to cover those modules better. The final results are shown in the column named *TUNING* in Table 6.5.

### 6.6.1 High-level Metrics Comparison

The framework set up for the experiments allowed a qualitative comparison of the different metrics. Test programs devised in the RT-level stage for maximizing the different metrics were compared with respect to other metrics such as the TA at gate-level and the final FC. This comparison is shown in Table 6.6, where we reported comparisons with the following metrics: Statement Coverage (SC), Branch Coverage (BC), Code Coverage (CC), Expression Coverage (EC), and Toggle Coverage (TC).

**Table 6.6.** High-level metrics comparison

| Program/metric | SC<br>[%] | BC<br>[%] | CC<br>[%] | EC<br>[%] | TC (RT)<br>[%] | TA<br>[%] | FC<br>[%] |
|---|---|---|---|---|---|---|---|
| SC | 98.49 | 85.49 | 90.00 | 69.77 | 83.91 | 37.52 | 41.92 |
| BC | 99.04 | 92.35 | 95.00 | 74.42 | 84.05 | 38.62 | 44.31 |
| CC | 99.04 | 92.61 | 97.50 | 74.42 | 84.05 | 39.73 | 44.81 |
| EC | 99.04 | 92.61 | 97.50 | 74.42 | 84.26 | 41.30 | 46.18 |
| TC (RT) | 99.04 | 92.61 | 97.50 | 74.42 | 90.32 | 42.04 | 46.28 |

The grayed cells in Table 6.6 represent the value on the metric that the set was intended to maximize. The last two columns contain the results regarding the TA at gate level and the FC reached by the same set of test programs.

First of all, it is interesting to notice the relationship between the different verification metrics, *e.g.*, maximizing the branch coverage leads to almost fully maximizing the SC. All code coverage metrics probably saturated: 99.04% SC, 92.61% branch coverage, 97.50% condition coverage and 74.42% expression coverage are probably the highest values attainable on the microprocessor description. It is possible to see that the expression coverage saturated earlier than the condition coverage. In contrast, the toggle coverage, the amount of signal that toggles from zero to one and one to zero, is not probably reaching its maximum.

Despite this fact, the final FC obtained by the different test sets is increasing, showing that a stable but smooth relationship exists when RT-level metrics are maximizing.

It should be remarked that, while the first four metrics are similarly fast to be calculated, the toggle coverage requires a more considerable effort. However, the toggle coverage is an interesting metric in subsequent stages in the design cycle, when the design is eventually synthesized to logic gates.

## 6.7 Conclusions

This chapter has shown and analyzed a methodology for collecting effective test-programs for microprocessor cores through the complete design cycle. The methodology employed is based on an almost automatic technique and could be applied to test-program generation for stand-alone microprocessor cores as well as the embedded ones in SOCs.

The main idea behind the test-program gathering process is to take advantage of all possible microprocessor descriptions delivered through the whole design cycle to generate suitable test programs able to attain a high FC% on gate-level descriptions. In fact, the proposed methodology's main efforts are on generation and analysis of test programs at the RT-level. However, the methodology presented is able to bring test programs together coming from the complete design life cycle. Thus, verification programs delivered before the manufacturing process can be used twice, for verification and for testing, in this way allowing the saving of time in the test-program generation process.

The proposed methodology is based on a step-by-step approach: test-program generation is performed each time a new microprocessor model is delivered. Moreover, being a cumulative process, which tackles high-level descriptions, the approach minimizes the use of explicit fault simulation to drive the test-program generation, greatly reducing computational efforts. In fact, fault simulation is only used at the end of the generation to complete the set of test programs.

In order to be applied, the approach requires that a wide set of metrics, mainly related to high-level descriptions, is available, and that the input stimuli generated when targeting these metrics are able to reach a significant coverage when fault-simulated on gate-level descriptions.

As a case study, the proposed methodology was implemented by tackling the PLASMA microprocessor. Along with this pipelined microprocessor, an ATPG tool called µGP was exploited, following the step-by-step method. The final test-set was composed of 15 test programs collected during the whole design process.

Finally, the results obtained were presented and analyzed using different tables that summarize the test-programs' performance in each design abstraction stage. Additionally, metrics comparison was performed by focusing on test programs generated at the RT-level, and the FC% they attain.

## Acknowledgments

We wish to thank Marco Giacomo Loggia and Massimiliano Schillaci for their crucial contributions.

## References

[1] Agrawal V, Bushnell M (2000) Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits. Norwell: Kluwer Academic Publishers

[2] Batcher K, Papachristou C (1999) Instruction randomization self test for processor cores. In: IEEE VLSI Test Symposium, 34-40

[3] Bieker U, Marwedel P (1995) Retargetable self-test program generation using constraint logic programming. In: 32nd ACM/IEEE Design Automation Conference, 605-611

[4] Chen L, Dey S  (2001) Software-based self-testing methodology for processor cores. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 20(3): 369-380

[5] Chen L, Dey S (2000) DEFUSE: a deterministic functional self-test methodology for processors. In: IEEE VLSI Test Symposium, 255-262

[6] Liu C-NJ, Chang C-Y, Jou J-Y, Lai M-C, Juan H-M (2000) A novel approach for functional coverage measurement in HDL circuits and systems. In: ISCAS2000: The 2000 IEEE International Symposium on Circuits and Systems, 217-220

[7] Corno F, Sanchez E, Sonza Reorda M, Squillero G (2004) Automatic test program generation – a case study. IEEE Design & Test, Special issue on Benchmarking for Design and Test, 21(2): 102-109

[8] Corno F, Sonza Reorda M, Squillero G (2003) Automatic test program generation for pipelined processors, In: SAC2003: The Eighteenth Annual ACM Symposium on Applied Computing, 736-740

[9] Corno F, Sonza Reorda M, Squillero G, Violante M (2001) On the test of microprocessor IP cores. In: IEEE Design, Automation & Test in Europe, 209-213

[10] Harman NA (2001) Verifying a simple pipelined microprocessor using Maude. In: Lecture Notes in Computer Science, vol 2267, 128-142

[11] Kranitis N, Paschalis A, Gizopoulos D, Zorian Y (2002) Effective software self-test methodology for processor cores. In: IEEE Design, Automation & Test in Europe, 592-597

[12] Kranitis N, Xenoulis G, Gizopoulos D, Paschalis A, Zorian Y (2003) Low-cost software-based self-testing of RISC processor cores. Computers and Digital Techniques, IEE Proceedings, 150(5): 355-60

[13] MIPS TECHNOLOGIES. (2002) MIPS32™ architecture for programmers volume I: introduction to the MIPS32™ architecture. Revision 1.90. http://www.mips.com

[14] Papachristou CA, Martin F, Nourani M (1999) Microprocessor based testing for core-based system on chip. In: ACM/IEEE Design Automation Conference, 586-591

[15] Parvathala P, Maneparambil K, Lindsay W (2002) FRITS – a microprocessor functional BIST method. In: IEEE International Test Conference, 590-598

[16] Paschalis A, Gizopoulos D (2004) Effective software-based self-test strategies for on-line periodic testing of embedded processors. In: Design, Automation and Test in Europe Conference and Exhibition, Volume: 1, 578-583

[17] Plasma CPU Model. http://www.opencores.org/projects/mips

[18] Rizk H, Papachristou C, Wolff F (2004) Designing self test programs for embedded DSP cores. In: IEEE Design Automation and Test in Europe Conference and Exhibition, 816-821

[19] Semiconductor Industry Association (2002) International Technology Roadmap for Semiconductors 2002 Update, http://www.semichips.org/pre_stat.cfm

[20] Shen J, Abraham J, Baker D, Hurson T, Kinkade M (1999) Functional verification of the Equator MAP1000 microprocessor. In: 36th ACM/IEEE Design Automation Conference, 169-174

[21] Shen J, Abraham JA (1998) Native mode functional test generation for processors with applications to self-test and design validation. In: IEEE International Test Conference, 990-999

[22] Thatte S, Abraham J (1980) Test generation for microprocessors. IEEE Transactions on Computers, C-29: 429-441

[23] Utamaphethai N, Blanton RD, Shen JP (1999) Superscalar processor validation at the microarchitecture level. In: 12th IEEE International Conference on VLSI Design, 300-305

[24] Van Campenhout D, Mudge TN, Hayes JP (1999) High-level test generation for design verification of pipelined microprocessors. In: ACM/IEEE Design Automation Conference, 185-188

[25] Velev MN, Bryant RE (2000) Formal verification of superscalar microprocessors with multicycle functional units, Exception, And Branch Prediction. In: ACM/IEEE Design Automation Conference, 112-117

# 7 Tackling Concurrency and Timing Problems

I. G. Harris

University of California Irvine, Department of Computer Science, USA

## 7.1 Abstract

Concurrent systems, either hardware or software, are notoriously difficult to design correctly in large part due to the complexities of nondeterministic execution. A concurrent system can perform many different correct computations for a given input sequence because the absolute order of execution is dependent on factors which cannot be known at design/compile time. Synchronization constructs are used to restrict the set of possible computations to correct computations only, but insertion of synchronization constructs is a manual and error-prone task. The detection of synchronization errors is made difficult because the manifestation of an error can depend on operation timing, which can change between executions. We define a class of synchronization errors and define the timing requirements to ensure the detection of these errors. We provide a coverage metric which can be used to determine whether or not a given test execution is sufficient to detect the defined class of synchronization errors.

## 7.2 Introduction

Timing is an important part of the correctness of hardware/software systems. Although timing correctness and functional correctness are often evaluated separately, they are equally important aspects of a system's behavior. The significance of timing derives from the applications which use hardware/software systems. Many applications are embedded controllers which perform a time-sensitive activity, such as an automatic braking system in a car in which a timing failure can be life threatening. Other applications include soft timing constraints, such as a video display system, which must process video frames at some minimum rate to maintain the illusion of continuous motion for the user. Systems like this may not be life critical, but timing failures result in a loss of output quality and a loss of revenue when potential customers purchase competing products.

We will loosely define timing correctness as the ability of a system to produce a result within a predefined time limit. This can be contrasted with functional correctness, which describes the ability of a system to produce a correct result, regardless of time. By this definition it is possible for a system to have correct timing but not correct function if an incorrect result is produced but it is produced on time. Strict timing goals have always been a part of the hardware design process, but the general software community traditionally considers performance as a secondary goal. Evidence for the low degree of importance placed on timing requirements in software is the lack of explicit timing constructs in software languages. Because timing in hardware must be well controlled, all hardware description languages have features which allow designers to explicitly state timing relationships between events. Embedded software, however, shares the need for strict timing requirements, in part because embedded software must interact directly with hardware. Hardware/software timing covalidation requires the use of techniques that can be applied in both the hardware and software domains.

Hardware/software systems are typically built from a number of concurrently executing processes, which must coordinate to complete system tasks. The degree of concurrency directly impacts system timing, and so it must be considered together with timing analysis. Concurrency is particularly relevant in the context of testing because systems containing concurrent execution are much more difficult to design than purely sequential systems. The difficulty stems largely from the fact that many different interleavings of concurrent operations are feasible for a given input sequence. This greatly increases the number of control flow possibilities, making it more difficult for a human to follow. As a result, the aspects of system design which involve concurrency are the source of a disproportionately large number of design errors. The importance of concurrency in hardware/software codesign and the difficulty of concurrent design make the detection of concurrency-related design errors a serious problem.

The main reason for the difficulty in analyzing timing and concurrency is the presence of nondeterminism in the execution sequence. That is, there may exist many correct instruction execution sequences for a given system input sequence. Nondeterminism is a useful design tool because it allows the designer to ignore detailed instruction sequences which do not impact functional correctness. Several major sources of nondeterminism exist in hardware/software systems.

*Operating System Scheduling* – The scheduling algorithms applied by most operating systems are nondeterministic because they employ runtime information that cannot be known prior to execution. This impacts the instruction sequence directly and also the performance. Scheduling algorithms are typically designed to optimize the average case schedule, but the variation in timing between schedules may be significant. In practice, this type of nondeterminism is sometimes avoided by foregoing the use of an operating system entirely and implementing the changes of control flow directly into the application processes.

*Microarchitectural Scheduling* – Microprocessors often use dynamic scheduling techniques, such as scoreboarding and Tomasulo's algorithm, to improve performance through instruction reordering. In practice, this type of nondeterminism may be avoided by using simple embedded processors that do not employ dynamic scheduling.

*Memory Hierarchy* – Using multiple levels of memory hierarchy makes memory delay nondeterministic because it depends on the hierarchy level at which the required data are found. Variable memory delays change timing but they do not directly impact instruction sequencing, although they may be used in scheduling. For example, an out-of-order processor may decide to delay the execution of an instruction because the data it requires may be in main memory rather than cache. It is possible to eliminate this type of nondeterminism by using only one level of memory hierarchy, but the performance will suffer. If the system can be designed to use no more memory than is contained in the L1 cache, then using only one level of hierarchy is feasible.

Nondeterminism is a problem for testing and validation because traditional (*i.e.* sequential) testing approaches assume that the correct execution sequence can be compared with a single known correct sequence to determine whether the system executed correctly under test. In the presence of nondeterminism, a system with a design error has the potential to produce an incorrect execution sequence, but there is no guarantee whether or not that will occur. As a result, it is entirely possible that a system with a design error could produce correct sequences during testing and produce an incorrect sequence later after the system is deployed for use. Another test problem associated with the existence of multiple, correct instruction sequences is that all correct sequences must be specified for comparison during test. This increases the memory required for testing, which is an issue in hardware testing, and it increases the time required to perform comparisons to check correctness.

Nondeterminism is managed in concurrent programs through the use of synchronization methods that restrict scheduling options to ensure correct operation. For example, if two processes cannot access some shared data at the same time, synchronization primitives must be added to the code to disallow the concurrent scheduling of operations which access the shared data. The task of using synchronization primitives in concurrent code is complicated and highly error prone. This chapter focuses on the errors involved in the synchronization between concurrent processes. We describe the most common methods of interprocess synchronization and then we describe the types of error that commonly occur and their effects on behavior. We discuss the detection requirements of synchronization errors and present a fault model which can be used as a coverage metric to indicate the ability of a given test sequence to detect synchronization errors.

## 7.3 Synchronization Techniques

Any model for concurrent computation must enable process interaction of two forms [1]:

- Contention – two processes competing for the same resource.
- Communication – two processes passing information from one to the other.

Both types of process interaction depend on the ability to perform synchronization. Synchronization can be defined as the task of limiting the allowable interleavings between the execution of multiple processes. For example, managing contention typically requires that the execution of critical sections of code in two processes is mutually exclusive. Ensuring mutual exclusion necessitates synchronization, because an interleaving of the processes should not be possible if it includes both processes executing their critical regions at the same time. Communication also requires synchronization, because the existence of communication implies a data dependency between processes, which cannot be violated. For example, if process X sends data to process Y, then the part of process Y which uses the data cannot execute until after process X has computed and sent the data.

Synchronization is accomplished by forcing processes to agree that a certain event has taken place. The occurrence of the event is used as a *synchronization point* around which the allowable interleavings can be constrained. There are several synchronization techniques used in hardware and software languages that are summarized here.

*Event synchronization* identifies some changes in system state (such as a signal changing value) to be an event that synchronizes a process. Events may come from outside of the system or from other processes. Event synchronization provides two primitives: the *wait* primitive, which causes a process to wait for an event, and the *post* primitive, which causes the event to occur. The placements of the wait and post primitives in each process defines the synchronization points. The wait primitive can be synchronous, which causes the invoking process to be blocked until the event occurs, or the wait may be asynchronous, which does not cause the process to block. In the asynchronous case, some type of event handler must be provided to be executed when the event does eventually occur. Event synchronization with synchronized wait is the common technique in hardware description languages such as Verilog and VHDL.

The *semaphore* technique [2] introduces two synchronization primitives called $P$ and $V$ which operate on natural numbers called semaphores. The semaphores are visible to all communicating processes, and events on semaphores are used for synchronization. $V(s)$ increments the value of $s$ while $P(s)$ tests the value of $s$ and decrements the value of $s$ if it is greater than zero. If the value of $s = 0$ then $P(s)$ will block, suspending the execution of the process invoking $P(s)$ until the value of $s > 0$. A key property of the $P$ and $V$ primitives is that they are atomic, meaning that once they are initiated they cannot be interrupted until they are complete. If multiple processes invoke $P$ or $V$ at the same time, then the executions occur sequentially in an arbitrary order. When a semaphore is incremented while there are several processes suspended by invoking $P(s)$ on the semaphore, the processes are chosen to complete the $P$ operation in an arbitrary order. Using semaphores, the incrementing and decrementing of semaphores are the events whose occurrence is agreed upon by all communicating processes. Possible interleavings are restricted by invoking $P$ in a process to suspend it until $V$ is invoked on the same semaphore by another process. The synchronization points are defined by the locations of the invocations of $P$ and $V$ in the processes.

A *monitor* [3,4] is an object whose access is limited to only one process at a time. All data inside the object is private and can only be accessed using the access functions of the class. Only one access function can be executed at a time. If a process attempts to access the monitor while it is being accessed, the process is suspended until the current access is complete. To accomplish synchronization, a monitor defines a set of condition variables. A wait operation is defined to cause a process to suspend until an event occurs on a condition variable, and a signal operation causes an event to occur on a condition variable. The locations of the invocations of the wait and signal operations are the synchronization points. The use of condition variables is similar to synchronous event synchronization.

Unlike the event, semaphore, and monitor synchronization methods, *message-based communication* does not assume that processes share memory. Instead, data are transferred between processes using the send and receive operations. It is well known, however, that message passing and shared memory communication schemes are equivalent. Any concurrent system implemented using one communication technique can also be implemented using the other. The send and receive operations can be either blocking or nonblocking, allowing the emulation of a range of synchronization methods. For example, the use of a nonblocking send and a blocking receive is equivalent to the synchronous event synchronization method used in most hardware description languages.

Another method of synchronization in a message-passing architecture is the use of *Remote Procedure Calls* (RPCs) [5], or the more general *rendezvous* [6]. An RPC enables a client process to invoke a function in another server process by using a procedure call which is similar to a normal procedure call within a single process. The client uses a send function to pass the name of the remote procedure to be invoked and the parameters of the procedure. The server process must invoke an accept function to indicate that it is ready to execute the requested procedure and the caller's send function must block until the request is accepted. Once the server process has completed the procedure it uses a return function to return the results to the caller. The caller must invoke a receive function to receive the function results from the server and the server's return function will block until the results are received. RPCs are asymmetric because the client can call procedures in the server, but the server cannot make requests of the client. Rendezvous is a generalized version of an RPC which allows processes to invoke procedures from each other in a symmetrical way.

## 7.4 A Class of Synchronization Errors

Each synchronization method requires the programmer to insert synchronization points manually into each process and this insertion process is a common source of errors in concurrent system design. We define this class of design errors and we describe the detection of errors in this class.

To understand synchronization errors it is necessary to establish the relationship between the placement of synchronization points in a system description and the behavior of that system. We will use the simple concurrent system depicted in Figure 7.1 to describe the impact of synchronization point placement on behavior. Figure 7.1 shows the outline of a system with two concurrent processes: a producer process and a consumer process. The code for the producer and consumer processes shown is minimal in order to highlight only the features of interest to this discussion. The producer sends data to the consumer through a variable *sh_data* which both processes share access to. The statement in the producer which assigns a value to the sh_data variable is referred to as a *definition* of sh_data, def(sh_data). The statement in the consumer which reads the value of sh_data is a *use* of sh_data, use(sh_data). In this example, event synchronization is assumed and the variable *ready* is used to indicate when the consumer is ready to receive new data. The synchronization point shown in the producer is the *wait* (*ready*) statement and the synchronization point in the consumer is the *ready* <= 1 statement.

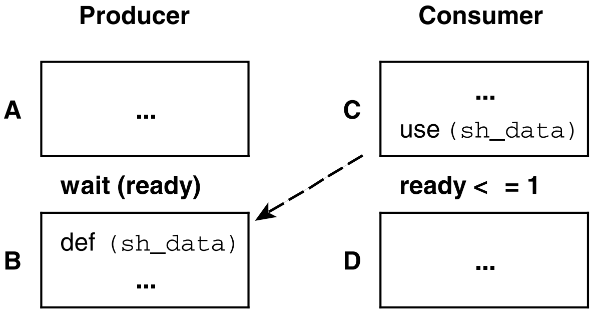


**Figure 7.1.** Synchronization in a producer/consumer example

The producer and consumer descriptions in Figure 7.1 are partitioned into four sequential blocks, A, B, C, and D, as determined by the placement of the synchronization points in the description. Each sequential block is a sequence of contiguous sequential instructions containing no synchronization points. The placement of the synchronization points establishes a dependency between sequential block B and C, so block B cannot execute until block C has completed. The dependency between B and C is needed to enforce the data dependency between the definition and use of the sh_data variable in blocks B and C. The definition and use of the sh_data variable represents a potential Write-After-Read (WAR) hazard, which is prevented using synchronization points.

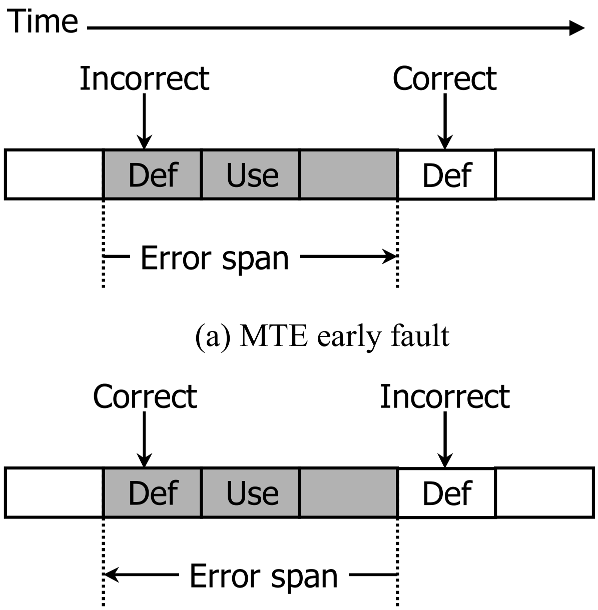
If the synchronization points are incorrectly placed, the sequential blocks are redefined and the WAR hazard may occur. If the wait statement in the producer is placed after the definition of sh_data, it is possible for the definition to occur before the use, causing the incorrect value of sh_data to be used. The same problem occurs if the *ready* <= 1 statement were accidentally placed before the use of the sh_data variable.

Incorrect placement of synchronization points can allow data dependency errors to exist between processes, but the manifestation of these errors depends on the scheduling. If the wait statement is misplaced, the definition of sh_data in the producer could be incorrectly scheduled before the use in the consumer and the error could be detected. However, it is possible that the wait could be misplaced but that the definition is never scheduled before the use during testing, and so the error is not detected. Such an undetected error could still manifest itself later in the product lifetime. This example demonstrates that the detection of synchronization errors depends on the schedule, which is in general nondeterministic.

## 7.5 A Fault Model for Synchronization Errors

To facilitate testing for synchronization errors a model is needed which enumerates all of the synchronization errors that will occur in an arbitrary design. The large number of potential design errors makes direct enumeration infeasible, so the model must be an abstract one. We refer to this model as a *fault model*, which defines a set of faults for an arbitrary design. Each fault described by the model represents a set of potential errors in a design and the detection of all faults ensures the detection of all errors of the type covered by the fault model.

According to their manifestation in time, design faults can be grouped into two classes: static faults, whose observation is independent of absolute event timing; and timing faults, whose observation depends on a specific timing of events on shared data. The observation of a static fault depends on the sequence of test pattern application, but not the absolute time of the application of each pattern. An example of a static fault is the replacement of the expression $x = y + 1$ with the incorrect expression $x = y + 2$, where the variable $x$ is shared between two processes. Once this fault is activated, its effects can be observed at any time before the signal $x$ is redefined. A timing fault exists when a signal is assigned to the correct value, but the event occurs at the incorrect time. A timing fault will cause a signal value to endure for the incorrect length of time. The timing fault effect can be observed only during the incorrect time period, which we will refer to as the *error span*. The difference between static faults and timing faults is that a timing fault is active during only a subset of the time period between two definitions, whereas a static fault is active during the entire time period between two definitions. Such a timing fault is a result of a synchronization error, because correct synchronization would prevent a process from using the value of $x$ until it has the correct value. Correct synchronization can be seen as a mechanism to make concurrent execution independent of timing. In the case of incorrect synchronization, the behavior of the system becomes timing dependent and timing faults can occur.



(a) MTE early fault

(b) MTE late fault

**Figure 7.2.** Two types of MTE fault

We refer to these faults as Mis-Timed Event (MTE) faults because they are caused by timing relationships between access to shared data. The MTE fault model [7] is derived from the all–definition–use pair metric [8] developed for the testing of sequential programs. An MTE fault is associated with each Definition–Use (DU) pair for each shared variable or object. The existence of an MTE fault indicates that the associated definition and use occur in the wrong sequence due to incorrect synchronization between the processes accessing the shared data. Two types of MTE fault can exist between a DU pair: $MTE_{early}$ indicates that the definition occurs earlier than the correct time, and $MTE_{late}$ indicates that the definition occurs later than the correct time. Figure 7.2a shows an $MTE_{early}$ fault whose error span extends forward in time from the incorrect time step, and Figure 7.2b shows an $MTE_{late}$ fault whose error span extends backward in time from the incorrect time step. Figure 7.2 assumes a discrete time model, which is common in hardware simulation, but the concept applies to continuous time as well.

### 7.5.1 Detection of Synchronization Faults

An MTE fault associated with a signal is detected only if there is a use of the shared variable inside the error span of the fault, as shown in Figure 7.2. The error span extends from the erroneous time step to the correct time step. Unfortunately, the precise position of the error span is not known, since simulation of the faulty circuit reveals only the erroneous time step. It is clear, however, that the error span must extend, either forward or backward in time, from the erroneous time step. In order to ensure that a use occurrence is within the error span of a fault, the use occurrence must be close to the corresponding definition occurrence in time. Also, a use occurrence must exist both earlier than the definition and later than the definition to detect both late and early MTE faults. These circumstances exist in Figure 7.2a and b where, in each case, the use occurrence is immediately adjacent to the erroneous time step. The detection of the $\text{MTE}_{\text{late}}$ fault is accomplished by the Use–Definition (UD) pair where the use occurs before the erroneous definition time step, and the $\text{MTE}_{\text{early}}$ fault is detected by the DU pair where the use occurs after the erroneous definition time step.

To ensure the detection of an MTE fault the associated DU or UD pair must be close in time. An *error span threshold* value *d* must be provided to define the maximum time difference between the definition and the use which is assumed to detect the fault. The error span threshold determines the sensitivity of the testing process to small perturbations in timing, so a small threshold ensures high sensitivity. However, if the threshold is too small then the timing behavior of the system may make the fault undetectable. For example, the operating system may impose a minimum delay to perform a context switch between two processes, and if the error span threshold is smaller than that minimum delay then MTE faults between the two processes will be undetectable. Identification of the minimum allowable threshold requires a solution to the minimum time separation problem [9], which is known to be NP-complete. We assume that the error span threshold is provided by a design/test engineer who has knowledge of the system timing behavior.

### 7.5.2 Fault Coverage Computation

The practical use of a fault model requires that there be an efficient procedure to determine which faults are detected when a given test sequence is applied to a design. We implemented MTE coverage computation using Verilog PLI, which allows MTE coverage to be computed for designs described in Verilog. The computation algorithm contains three main steps.

1. *DU/UD pairs identification.* In this step, we generate lists of DU/UD pairs for each signal by analyzing the behavioral description. Since a signal can be defined and used in multiple modules in a hierarchical design, it may have different names in different modules. In order to catch any occurrence of the signal, we first find the top module in which the signal is first declared as a signal. Along the connection down to the submodules we recursively find all modules

using the signal. In each such a module, we locate the definition and use statements and register a callback function for each occurrence statement. After locating all definition and use occurrence, all DU pairs sharing the same use are associated with that use occurrence and all UD pairs sharing the same definition are associated with that definition occurrence. The initial value of time separation for each DU/UD pair is set to a negative number, which is updated during the simulation.

2. *Simulation.* The behavioral description is simulated with the test sequence. For each signal there is a record of the current definition which defines the current value of the signal. The record includes the location of the current definition and the time step when the definition occurred. The record of the current definition is updated every time a new definition occurs. During the simulation, a callback function is called when a statement with definition or use occurrence is executed. The callback function then records the time step of the occurrence. For a definition occurrence, the callback function updates the record of the current definition and calculates the new value of time separation for each UD pair associated with the current definition. If the new value of separation is less than the old one, the callback function updates the time separation of the UD pair. For a use occurrence, the callback function calculates the value of time separation for the DU pair from the current definition to the use and updates the value of separation if the new value is smaller.
3. *MTE fault coverage analysis.* In this step, the simulation results are analyzed and the MTE fault coverage is calculated for a range of thresholds. For a given threshold, the MTE fault coverage is the ratio of the number of the DU/UD pairs executed within threshold to the number of all DU/UD pairs. Since the value of the threshold strongly affects the fault coverage, the coverage result is presented by a curve rather than a number. The resulting curve shows the trend of coverage over a range of threshold values.

## 7.6 Experimental Results

We experimented with the MTE fault analysis tool using four industry designs that were provided to us with functional test sequences which we evaluated for MTE coverage. Each benchmark was provided in Verilog and the Cadence Verilog-XL simulator was used to gather coverage information. The first benchmark is an implementation of type 1 ATM Adaptation Layer protocol that abstracts the ATM layer from higher level communication protocols. The type 1 AAL protocol is used to provide a Constant Bit Rate (CBR) service such as conventional voice service and existing leased line service. The second benchmark is an implementation of a four-ports data switch that contains an arbiter and four ports to receive and send data. Each port sends requests for the internal bus and the arbiter chooses one to allow access. The third benchmark is an implementation of Dual Tone Multi-Frequency (DTMF) receiver. DTMF signals are the control tones generated by standard touch-tone telephones. Pressing a key causes the telephone to generate a pair of tones, one from the high-frequency group, and one from the low-frequency group. To detect the tones, the DTMF receiver utilizes Goertzel's algorithm to calculate the frequency response at the DTMF center frequencies. Once calculated, the frequency response is analyzed to determine which DTMF digit was found. The fourth benchmark is a simple RISC CPU core.

**Table 7.7.** Benchmark information and coverage summary

| Benchmark | Lines | Blocks | Signals | Pairs | Stmt | MTE |
|---|---|---|---|---|---|---|
| AAL1 | 2068 | 54 | 22 | 1732 | 0.70 | 0.10 |
| Switch | 1269 | 28 | 29 | 200 | 0.93 | 0.65 |
| Risc8 | 2302 | 50 | 37 | 1032 | 0.60 | 0.54 |
| DTMF | 8383 | 77 | 17 | 262 | 0.54 | 0.46 |

Table 7.1 summarizes the benchmark information and the coverage results. The first four columns in order show the benchmark names, lines of Verilog code, number of *always* (concurrent) blocks, and number of signals used. The number of signals is relevant because each signal acts as a shared variable and the MTE faults are associated with definitions and uses of the signals. The fifth column contains the number of DU/UD pairs for all of the signals, which is also the number of MTE faults in the design. The sixth column shows the statement coverage produced during test application, the fraction of statements covered during simulation. The seventh column shows the maximum MTE coverage values, the fraction of MTE faults which are detected by the test sequence. The MTE coverages reported in Table 7.1 are maximal because the error span threshold was set to the maximum value.

Detailed MTE coverage results for the AAL1 benchmark are shown in Figures 7.3 and 7.4. Figure 7.3 shows the variation in MTE coverage over a range of error threshold values. The MTE coverage rises as the error threshold increases, because larger separation between DU/UD pairs is allowed. The maximum MTE coverage in this example is quite low, only 0.10, and MTE coverage is used to identify weaknesses in the test sequence and locate coverage holes. Upon examining the benchmark, we find that 87% of the DU/UD pairs are associated with one signal rec_seq that has 28 definitions and 27 uses. This results in 1512 DU/UD pairs.

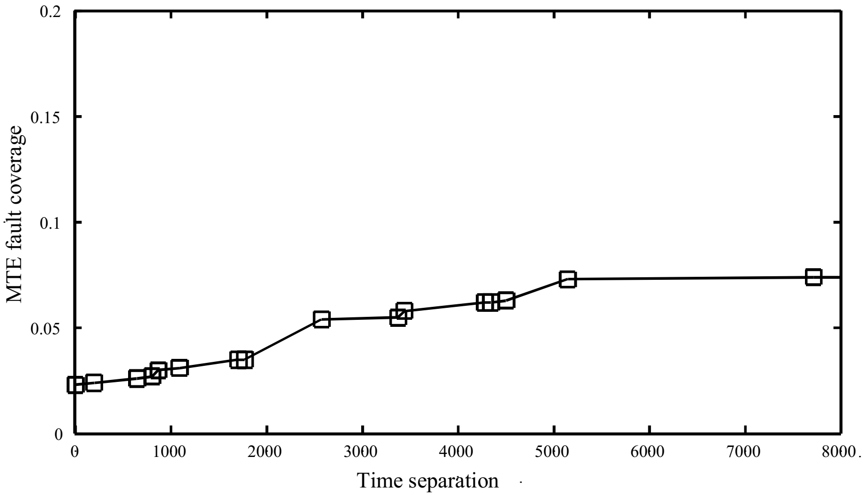


**Figure 7.3.** AAL1 MTE fault coverage distribution

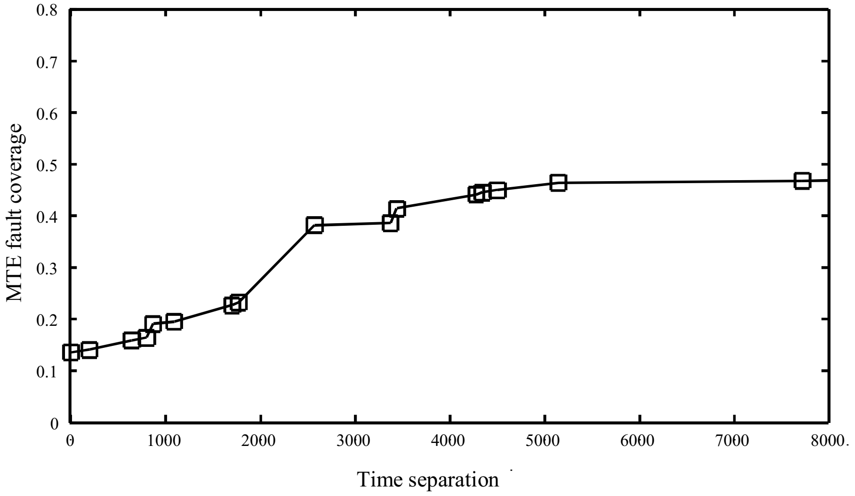


**Figure 7.4.** AAL MTE coverage without the rec_seq signal

However, only 35 out of these 1512 pairs were executed during simulation. The rec_seq signal is defined 28 times in the *receiver.rec_CPU.chk1* module, whose statement coverage is only 0.22, and is used 23 times in the *receiver.rec_cpu.fsm1* module, whose statement coverage is 0.40. Therefore, most of the definitions and uses are not executed and the MTE fault coverage of this signal is only 0.02, which results in the overall low MTE coverage on the design. Figure 7.4 shows the distribution of MTE coverage without consideration of signal rec_seq and the coverage increases to 0.63. The test sequence needs to be enhanced to cover the DU/UD pairs involving uses in the *receiver.rec_cpu.fsm* module. In this case the MTE fault model identifies a weakness in the test sequence and provides direction on how the sequence should be changed to improve the completeness of testing.

## 7.7 Conclusions

As hardware/software codesign is increasing applied for the design of embedded applications, both functional and timing correctness of these designs becomes more important. Process synchronization is manually intensive and has proven to be a difficult task due to the complex interprocess dependencies that must be considered. Many synchronization methods are in use, which attempt to ease the synchronization task by providing abstract primitives for use by the designer. However, it is not possible to enable efficient design while completely hiding the intricacies of the synchronization task from the designer. As a result, synchronization is likely to be a difficult and error-prone process for the foreseeable future. A significant body of research is dedicated to hardware and software validation [10], but little of this existing research focuses on the synchronization problem. Given the inherent difficulty in synchronization, further research in this area can be expected.

## Acknowledgments

Work on this chapter was supported in part by the National Science Foundation under grant number 0204134

## References

[1] Ben-Ari M (1990) Principles of concurrent and distributed programming. Prentice Hall International (UK) Ltd

[2] Dijkstra EW (1968) Cooperating sequential processes, programming languages, 43-112

[3] Hoare CAR (1974) Monitors: an operating system structuring concept. Communications of the ACM 17(10): 549-557

[4] Brinch Hansen P (1973) Operating system principles. Prentice Hall, Englewood Cliffs, NJ

[5] Brinch Hansen P (1978) Distributed processes: a concurrent programming concept. Communications of the ACM 21: 934-941

[6] Hoare CAR (1978) Communicating sequential processes. Communications of the ACM 21: 666-667

[7] Zhang Q, Harris IG (2001) A Validation fault model for timing-induced functional errors. In: International Test Conference, 813-820

[8] Rapps S, Weyuker EJ (1985) Selecting software test data using data flow information. IEEE Transactions on Software Engineering SE-11(4): 367-375

[9] Chakraborty S, Dill DL (1997) Approximate algorithms for time separation of events. In International Conference on Computer-Aided Design, 190-198

[10] Harris IG (2003) Fault models and test generation for hardware-software covalidation. IEEE Design and Test of Computers 20(4): 40-47

# 8 An Approach to System-level Design for Test

G. Jervan, R. Ubar, Z. Peng, P. Eles

Linköping University, Linköping, Sweden
Tallinn University of Technology, Tallinn, Estonia

## 8.1 Abstract

In this chapter we will describe a Design-for-Test (DfT) methodology for systems-on-chip. We have developed a hybrid Built-In Self-Test (BIST) approach, where the test set is assembled from pseudorandom test patterns that are generated online and deterministic test patterns that are generated offline and stored in the system. We have analyzed the aspects related to the cost calculation of such a hybrid BIST approach and will propose a test cost minimization strategy for single-core designs. We have then extended the same approach for multi-core designs and developed a test time minimization methodology under tester memory constraints. We will demonstrate the applicability and efficiency of the proposed approach for cores with different core-level DfT structures and for systems with different system-level test architectures.

## 8.2 Introduction

The rapid advances of the microelectronics technology in recent years have brought new possibilities to integrated circuits (ICs) design and manufacturing. Many systems are nowadays designed by embedding predesigned and preverified complex functional blocks, usually referred to as cores, into one single die. While this core-based design technique has led to increased design productivity, it introduces additional test-related problems, which are due to, among others, intellectual property protection. These additional testing problems, together with the test problems induced by the complexity and heterogeneous nature of System-On-Chip (SOC), pose great challenges to the SOC testing community.

It should first be noted that, even though the core-based design strategy is, to a certain extent, similar to traditional system-on-board (SOB) design, where individual chips are designed and then integrated into a board, production tests of SOC and of SOB are very different. In SOB testing, the individual chips are manufactured and tested first before they are integrated into the board. The individual SOC cores, while predesign and preverified, will not be tested until they are integrated into a system chip. Therefore, a core is not tested individually, but rather as a part of the overall system chip test. This means that the divide-and-conquer testing strategy traditionally used to deal with the complexity of testing a complex board cannot be applied directly in SOC testing.

Besides the increased complexity, the difficulty of SOC testing is due to its heterogeneous nature. Typically, a SOC consists of microprocessor cores, digital logic blocks, analog devices, and memory structures. These different types of component were traditionally tested, as separate chips, by dedicated automatic test equipment of different types. Now they must be tested all together as a single chip, either by a super tester, which is capable of handling the different types of cores and is very expensive, or by multiple testers, which is very time consuming due to the handling time of moving from one tester to another.

Another problem related to testing embedded cores as a part of system test is due to the limited knowledge the system integrator has about the internal structure of a core. This may be due to intellectual property protection or the use of complex hard cores. In this situation, the core developer will provide the test patterns and insert Design-for-Test (DfT) mechanisms into the core. Since the core developer has no idea about the overall SOC design and test strategy to be used, the inserted DfT mechanism may not be compatible with the overall design and test philosophy, leading usually to low test quality or high overhead. This problem needs to be solved in order to guarantee the high quality level of SOC products.

Another key issue to be addressed for SOC testing is the implementation of test access mechanisms on chip. For traditional SOB design, direct test access to the peripheries of the basic components, in the form of separate chips, is usually available. For the corresponding cores embedded deeply in a SOC, such access is impossible. Therefore, additional test access mechanisms must be included in a SOC to connect the core peripheries to the test sources and sinks, which are the SOC pins when testing by an external tester is assumed.

The design of the test access mechanism must be considered together with the test-scheduling problem, in order to reduce the silicon area used for test access and to minimize the total test application time, which includes the time to test the individual cores and user-defined logic as well as the time to test their interconnections. The issue of power dissipation in test mode should also be considered in order to prevent the chip being damaged by overheating during test. Since the problems of test access mechanism design, test scheduling, test application time minimization, and test power consideration are interdependent, they must be solved together in an integrated design environment.

Many of the testing problems discussed above can be overcome by using a Built-In Self-Test (BIST) strategy. For example, the test access cost can be substantially reduced by putting the test sources and sinks next to the cores to be tested. BIST can also be used to deal with the discrepancy between the speed of the SOC, which is increasing rapidly, and that of the tester, which will soon be too slow to match typical SOC clock frequencies. The introduction of BIST mechanisms in a SOC will also improve the diagnosis ability and field-test capability, which are essential for many applications where regular operation and maintenance testing is needed.

Since the introduction of BIST mechanisms into a SOC is a complex task, we need to develop powerful automated design methods and tools to optimize the test function, together with the other design criteria and to speed up the design process. Such methods and tools are collectively called DfT methods. In this chapter we are going to concentrate on one of those methods, namely BIST. We will describe hybrid BIST methodology as an improvement of a classical BIST approach, and describe optimization methods for such architectures.

A classical BIST architecture consists of a Test Pattern Generator (TPG), a Test Response Analyzer (TRA) and a BIST Control Unit (BCU), all implemented on the chip. Different implementations of such BIST architectures have been available, and some of them have got wide acceptance. Unfortunately, the classical BIST approaches suffer the problems of inducing additional delay to the circuitry and requiring a relatively long test application time.

In particular, one major problem of the classical BIST implementation is due to the fact that the TPG for BIST is implemented by linear feedback shift registers (LFSRs) [1], [2], [20]. Since the test patterns generated by an LFSR are pseudorandom by nature and have linear dependencies [6], the LFSR-based approach often does not guarantee a sufficiently high fault coverage (especially in the case of large and complex designs), and demands very long test application times in addition to high area overheads. Therefore, several proposals have been made to combine pseudorandom test patterns, generated by LFSRs, with deterministic patterns [4], [7], [8], [14], [15], [21], to form a hybrid BIST solution.

The main concern of the hybrid BIST approaches has been to improve the fault coverage by mixing pseudorandom vectors with deterministic ones, while the issue of test cost minimization has not been addressed directly.

In the following sections, we will analyze first the aspects related to the cost calculation of hybrid BIST. We will explain the basic concepts based on single-core designs. Thereafter, we will demonstrate how those concepts can be expanded for multi-core designs.

## 8.3 Hybrid Built-in Self-test

As described earlier, a typical self-test approach usually employs some form of pseudorandom TPGs. These test sequences are often very long and not sufficient to detect all the faults. To avoid the test quality loss due to random pattern-resistant faults and to speed up the testing process, we can apply deterministic test patterns targeting the random resistant and difficult-to-test faults. Such a hybrid BIST approach usually starts with a pseudorandom test sequence of length $L$. After the application of pseudorandom patterns, a stored test approach will be used [9]. For the stored test approach, pre-computed test patterns are applied to the core under test in order to reach the desirable fault coverage level. For offline generation of the deterministic test patterns, arbitrary software test generators may be used based on deterministic, random or genetic algorithms.

In a hybrid BIST technique the length of the pseudorandom test is an important design parameter, as it determines the behavior of the whole test process. A shorter pseudorandom test sequence implies a larger deterministic test set. This requires additional memory space, but at the same time it shortens the overall test time. A longer pseudorandom test, on the other hand, will lead to larger test application time with reduced memory requirement. Therefore, it is crucial to determine the optimal length of pseudorandom test in order to minimize the total testing cost.

Figure 8.1 illustrates graphically the total cost of a hybrid BIST consisting of pseudorandom test patterns and stored test patterns generated offline. The horizontal axis in Figure 8.1 denotes the fault coverage achieved by the pseudorandom test sequence before switching from the pseudorandom test to the stored test. Zero fault coverage is the case when only stored test patterns are used and, therefore, the cost of the stored test is greatest at this point. The figure illustrates the situation where 100% fault coverage is achievable with pseudorandom vectors alone.

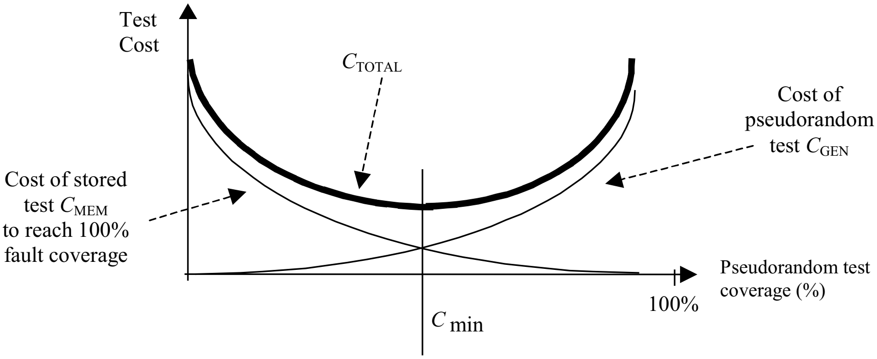


**Figure 8.1.** Cost calculation for hybrid BIST (under 100% assumption)

The total test cost of the hybrid BIST $C_{\text{TOTAL}}$ can therefore be defined as

$$C_{\text{TOTAL}} = C_{\text{GEN}} + C_{\text{MEM}} = \alpha L + \beta S \qquad (8.1)$$

where $C_{\text{GEN}}$ is the cost related to the effort for generating $L$ pseudorandom test patterns (number of clock cycles), $C_{\text{MEM}}$ is related to the memory cost for storing $S$ pre-computed test patterns to improve the pseudorandom test set, and $\alpha$, $\beta$ are constants to map the test length and memory space to the costs of the two parts of the test solutions.

We should note that defining the test cost as a sum of two costs, the cost of time for the pseudorandom test generation and the cost of memory associated with storing the TPG-produced test, is a rather simplified cost model for the hybrid BIST technique. In this simplified model, neither the basic cost of memory (or its equivalent) occupied by an LFSR-based generator, nor the time needed for generating deterministic test patterns are taken into account. However, these aspects can easily be added to the cost calculation formula after the desired hardware architecture is chosen. In this chapter we are going to provide the algorithms to find the best trade-off between the length of pseudorandom test sequence and the number of deterministic patterns. For making such a trade-off, the basic implementation costs are invariant and will not influence the optimal selection of the hybrid BIST parameters.

On the other hand, the attempt to add "time" to "space" (even in terms of their cost) seems rather controversial, as it is very hard to specify which one costs more in general (or even in particular cases) and how to estimate these costs. This was also the reason why the total cost of the BIST function is not considered in this chapter. The values of parameters $\alpha$ and $\beta$ in the cost function are left to be determined by the designer and can be seen as one of the design decisions. If needed, it is possible to separate these two different costs (time and space), and consider, for example, one of them as a design constraint.

Figure 8.1 illustrates also how the cost of pseudorandom testing is increasing when striving to higher fault coverage (the $C_{\text{GEN}}$ curve). In general, it can be very expensive to achieve high fault coverage with pseudorandom test patterns alone. The $C_{\text{MEM}}$ curve describes the cost that we have to pay for storing additional precomputed tests at the given fault coverage level reached by pseudorandom testing. The total cost $C_{\text{TOTAL}}$ is the sum of the above two costs. The $C_{\text{TOTAL}}$ curve is illustrated in Figure 8.1, where the minimum point is marked as $C_{\text{min}}$.

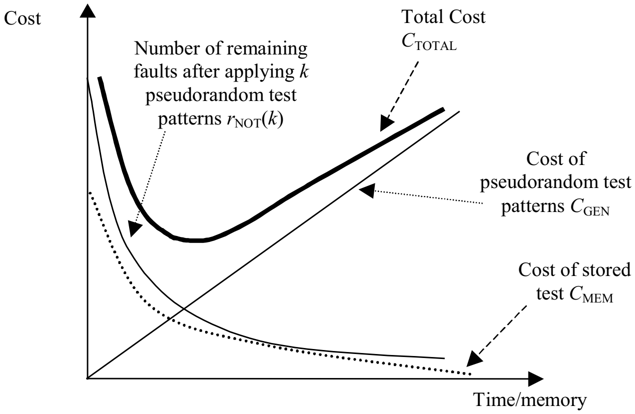


**Figure 8.2.** Cost calculation for hybrid BIST

As mentioned earlier, in many situations 100% fault coverage is not achievable with only pseudorandom vectors. Therefore, we have to include this assumption in the total cost calculation. The situation is illustrated in Figure 8.2, where the horizontal axis indicates the number of pseudorandom patterns applied, instead of the fault coverage level. The curve of the total cost $C_{\text{TOTAL}}$ is still the sum of two cost curves $C_{\text{GEN}} + C_{\text{MEM}}$, with the new assumption that the maximum fault coverage is achievable only by either the hybrid BIST or a pure deterministic test.

### 8.3.1 Hybrid Built-in Self-test Cost Optimization

In the following we will describe different methods to find the global minimum of the Total Cost curve, while testing every core in isolation. Creating the curve $C_{\mathrm{GEN}} = \alpha L$ is not difficult. For this purpose, only a simulation of the behavior of the LSFR used for pseudorandom test pattern generation is needed. A fault simulation should be carried out for the complete test sequence generated by the LFSR. As a result of such a simulation, we find for each clock cycle the list of faults which are covered at this clock cycle.

As an example, in Table 8.1 a fragment of the results of BIST simulation for the ISCAS'85 circuit c880 [3] is given, where

- *k* denotes the number of the clock cycle,
- $r_{\mathrm{DET}}(k)$ is the number of new faults detected (covered) by the test pattern generated at the clock signal *k*,
- $r_{\mathrm{NOT}}(k)$ is the number of remaining faults after applying the sequence of patterns generated by the *k* clock signals,
- FC(*k*) is the fault coverage reached by the sequence of patterns generated by the *k* clock signals

**Table 8.1.** Pseudorandom test results

| *k* | $r_{\mathrm{DET}}(k)$ | $r_{\mathrm{NOT}}(k)$ | FC(*k*) | *k* | $r_{\mathrm{DET}}(k)$ | $r_{\mathrm{NOT}}(k)$ | FC(*k*) |
|---|---|---|---|---|---|---|---|
| 0 | 155 | 839 | 15.593561% | 148 | 13 | 132 | 86.720322% |
| 1 | 76 | 763 | 23.239437% | 200 | 18 | 114 | 88.531189% |
| 2 | 65 | 698 | 29.778671% | 322 | 13 | 101 | 89.839035% |
| 3 | 90 | 608 | 38.832996% | 411 | 31 | 70 | 92.957748% |
| 4 | 44 | 564 | 43.259556% | 707 | 24 | 46 | 95.372231% |
| 5 | 39 | 525 | 47.183098% | 954 | 18 | 28 | 97.183098% |
| 10 | 104 | 421 | 57.645874% | 1535 | 4 | 24 | 97.585510% |
| 15 | 66 | 355 | 64.285713% | 1560 | 8 | 16 | 98.390343% |
| 20 | 44 | 311 | 68.712273% | 2153 | 11 | 5 | 99.496979% |
| 28 | 42 | 269 | 72.937622% | 3449 | 2 | 3 | 99.698189% |
| 50 | 51 | 218 | 78.068413% | 4519 | 2 | 1 | 99.899399% |
| 70 | 57 | 161 | 83.802818% | 4520 | 1 | 0 | 100.000000% |
| 100 | 16 | 145 | 85.412476% | | | | |

In the list of BIST simulation results, not all clock cycles should be presented. We are only interested in the clock numbers at which at least one new fault will be covered, and the total fault coverage for the pseudorandom test sequence up to this clock number increases. Let us call such clock numbers and the corresponding pseudorandom test patterns *efficient clocks* and *efficient patterns*. The rows in Table 8.1 correspond to the efficient, not all, clocks for the circuit c880.

If we decide to switch from pseudorandom mode to the deterministic mode after the clock number *k*, then *L* = *k*.

More difficult is to find the values for $C_{\mathrm{MEM}} = \beta S$. Let *t*(*k*) be the number of test patterns needed to cover $r_{\mathrm{NOT}}(k)$ not yet detected faults (these patterns should be pre-computed and used as stored test patterns in the hybrid BIST). As an example, these data for the circuit c880 are depicted in Table 8.2. Calculation of the data in the column $t(k)$ of Table 8.2 is the most expensive procedure. In the following section the difficulties and possible ways to solve the problem are discussed.

**Table 8.2.** Automatic TPG (ATPG) results

| $k$ | $t(k)$ | $k$ | $t(k)$ |
|---|---|---|---|
| 0 | 104 | 148 | 46 |
| 1 | 104 | 200 | 41 |
| 2 | 100 | 322 | 35 |
| 3 | 101 | 411 | 26 |
| 4 | 99 | 707 | 17 |
| 5 | 99 | 954 | 12 |
| 10 | 95 | 1535 | 11 |
| 15 | 92 | 1560 | 7 |
| 20 | 87 | 2153 | 3 |
| 28 | 81 | 3449 | 2 |
| 50 | 74 | 4519 | 1 |
| 70 | 58 | 4520 | 0 |
| 100 | 52 | | |

### *The Optimization Algorithms*

There are two approaches to find $t(k)$: ATPG based and fault-table based. Let us have the following notations:

- $i$ – the current number of the entry in the tables for PRG and ATPG;
- $k(i)$ – the number of the clock cycle of the efficient clock $i$;
- $R_{\mathrm{DET}}(i)$ – the set of new faults detected (covered) by the pseudorandom test pattern which is generated at the efficient clock signal number $i$;
- $R_{\mathrm{NOT}}(i)$ – the set of not-yet-covered faults after applying the pseudorandom test pattern number $i$;
- $T(i)$ – the set of test patterns needed and found by the ATPG to cover the faults in $R_{\mathrm{NOT}}(i)$;
- $N$ – the number of all efficient patterns in the sequence created by the pseudorandom test;
- FT – the fault table for a given set of tests $T$ and for the given set of faults $R$: the element $\varepsilon_{jk}$ ($j$ is ranged over the test set $T$, and $k$ over the fault set $R$) in the table is defined as $\varepsilon_{jk} = 1$ if the test $t_j \in T$ detects the $r_k \in R$; otherwise $\varepsilon_{jk} = 0$.

***Algorithm 1:*** ATPG-based approach for finding test sets $T(i)$

```
1. Let $q := N$;
2. Generate for $R_{\mathrm{NOT}}(q)$ a test set $T(q)$, $T := T(q)$, $t(q) := |T(q)|$;
3. For all $q = N - 1, N - 2, \ldots 1$:
   Generate for the faults $R_{\mathrm{NOT}}(q)$ not covered by test $T$ a test set $T(q)$,
   $T := T + T(q)$, $t(q) := |T|$.
```

The above algorithm generates a new deterministic test set for the not-yet-detected faults at every efficient clock cycle. In this way we have the complete test set (consisting of pseudorandom and deterministic test vectors) for every efficient clock, which can reach to the maximal achievable fault coverage. The number of deterministic test vectors at all efficient clocks is then used to create the curve $C_{\mathrm{MEM}}(\beta S)$. The algorithm is straightforward; however, it is very time consuming because of the repetitive use of ATPG.

***Algorithm 2:*** Fault-table-based approach for finding test sets $T(i)$

```
1. Let $q = 1$; calculate the test $T(q)$ for the whole set of faults $R$, create the fault
   table FT;
2. For all $q = 2, 3, \ldots, N$:
      Create a new fault table FT by removing from it the faults $R_{\mathrm{DET}}(q-1)$,
      and optimize the test set $T(q-1)$ in relation to the new FT. The optimized
      test set is $T(q)$.
```

This algorithm starts by generating a test set $T$ for all detectable faults. Based on the fault simulation results, a fault table FT will be created. By applying $k$ pseudorandom patterns we can remove all faults from the original fault table, which were covered by the pseudorandom vectors and by using static test compaction reduce the original deterministic test set. These modifications should be performed iteratively for all possible breakpoints to calculate the curve $C_{\mathrm{MEM}}(\beta S)$ and to use this information to find the optimal $C_{\mathrm{TOTAL}}$. More details about the algorithms can be found in [17].

The main limitation of the above algorithms is that, in the case of very large circuits, both of them are very time consuming and not applicable in the case of large and complex designs. It would be desirable, therefore, to find a solution that is close to the optimum of the total cost curve by only few sampled calculations of the total cost for selected values of $i$, $1 \le i \le N$.

For this purpose a fast estimation algorithm to search the close to optimum solution by using just a few samples from the whole test generation experiments set can be used. As available data for such a kind of estimation, the number of not-yet-covered faults in $R_{\mathrm{NOT}}(k)$ can be used. The value of $R_{\mathrm{NOT}}(k)$ can be acquired directly from the PRG simulation results and is available for every significant time moment (see Table 8.1). Based on the value of $|R_{\mathrm{NOT}}(k)|$ it is possible to reason about the expected number of test patterns needed for covering the faults in $R_{\mathrm{NOT}}(k)$, and this information can be used to estimate the total cost of the hybrid BIST solution [9], [11].

The estimation procedure does not provide, in most cases, the final solution, but can be used as a good starting point to search for the close to the optimum solution by sampled calculation of the real cost. Estimated cost would suggest the first solution for dividing the BIST into two parts (PRG and deterministic based). Based on this starting point, the exploration can be carried out by using some classical methods, like Tabu search [5] or simulated annealing [12]. The results are presented in Table 8.3 [10], [18].

Experiments were carried out on the ISCAS’85 benchmark circuits for comparing Algorithms 1 and 2, and for investigating the efficiency of the Tabu method for optimizing the hybrid BIST. The Turbo Tester toolset [16] was used for deterministic TPG, fault simulation, and test set compaction.

In the columns of Table 8.3 the following data are depicted: ISCAS’85 benchmark circuit name; $L_{PR}$, the length of the pseudorandom test sequence; FC, the fault coverage; $S_{DET}$, the number of deterministic test patterns, generated by the ATPG, without using any pseudorandom patterns; $T_1$ and $T_2$, the time (seconds) needed for calculating the cost curve by Algorithms 1 and 2; $T_3$, the time (seconds) to find the optimal cost by using Tabu search; Acc, the accuracy of the Tabu search solution as a percentage compared with the exact solution found from the exact cost curve (Algorithm 1). In the hybrid BIST column we have presented the optimized hybrid BIST solution. $L$ is a number of pseudorandom patterns and $S$ is a number of deterministic patterns in the optimized hybrid BIST sequence. The hybrid BIST cost has been optimized based on assumption that one clock cycle equals one byte of memory. As can be seen, in a typical case less than half of the deterministic vectors and only a small fraction of pseudorandom vectors are needed; however, the maximum achievable fault coverage is guaranteed and achieved. Obviously, by changing the ratio of the cost parameters (time and memory) we will drive the optimization algorithm to the different solutions, *e.g.*, longer pseudorandom sequence and less deterministic patterns or *vice versa*.

**Table 8.3.** Experimental results of creating optimized hybrid BIST

| Circuit | Pseudorandom test | | Stored test | | Hybrid BIST | | Calculation cost | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $L_{PR}$ | FC | $S_{DET}$ | $C$ | $L$ | $S$ | $T_1$ | $T_2$ | $T_3$ | Acc |
| C432 | 780 | 93.0 | 80 | 93.0 | 91 | 21 | 1632 | 21 | 2.85 | 100.0 |
| C499 | 2036 | 99.3 | 132 | 99.3 | 78 | 60 | 74 | 3 | 0.50 | 100.0 |
| C880 | 5589 | 100.0 | 77 | 100.0 | 121 | 48 | 17 | 2 | 0.26 | 99.7 |
| C1355 | 1522 | 99.5 | 126 | 99.5 | 121 | 52 | 133 | 5 | 0.83 | 99.5 |
| C1908 | 5803 | 99.5 | 143 | 99.5 | 105 | 123 | 2132 | 25 | 3.83 | 100.0 |
| C2670 | 6581 | 84.9 | 155 | 99.5 | 444 | 77 | 230 | 13 | 0.99 | 99.1 |
| C3540 | 8734 | 95.5 | 211 | 95.5 | 297 | 110 | 22601 | 122 | 7.37 | 100.0 |
| C5315 | 2318 | 98.9 | 171 | 98.9 | 711 | 12 | 2593 | 38 | 1.81 | 97.2 |
| C6288 | 210 | 99.3 | 45 | 99.3 | 20 | 20 | 200 | 6 | 1.70 | 100.0 |
| C7552 | 18704 | 93.7 | 267 | 97.1 | 583 | 61 | 15004 | 129 | 3.70 | 99.7 |

## 8.4 Hybrid Built-in Self-test for Multi-core Systems

In the previous section we described the basic principles of hybrid BIST and discussed methods for test cost calculation and optimization for individual cores in isolation. In this section we concentrate on hybrid BIST optimization for multi-core designs. As total cost minimization for multi-core systems is an extremely complex problem and is rarely used in reality, the main emphasis here is on test time minimization under memory constraints with different test architectures. The memory constraints can be seen as limitations of on-chip memory or automatic test equipment, where the deterministic test set will be stored, and therefore with high practical importance. We will concentrate on two large classes of test architectures. In one case we assume that every core is equipped with its own pseudorandom pattern generator and only deterministic patterns have to be transported to the cores. In the second case we assume test pattern broadcasting, where both pseudorandom and deterministic test patterns have to be transported to the cores under test. For both architectures we will describe test-per-clock as well as test-per-scan application schemes.

It is important to mention here that the following approaches do not take into account test power, nor do we propose any methods for test access mechanism optimization. Those problems can be solved after the efficient test set for every individual core has been developed and, therefore, are not considered here.

### 8.4.1 Built-in Self-test Time Minimization for Systems with Independent Built-in Self-test Resources

We start with a test architecture where every core has its own dedicated BIST logic that is capable producing a set of independent pseudorandom test patterns, *i.e.*, the pseudorandom test sets for all the cores can be carried out simultaneously.

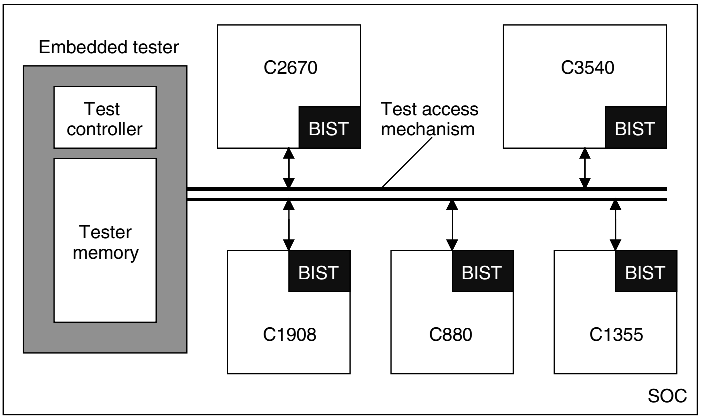
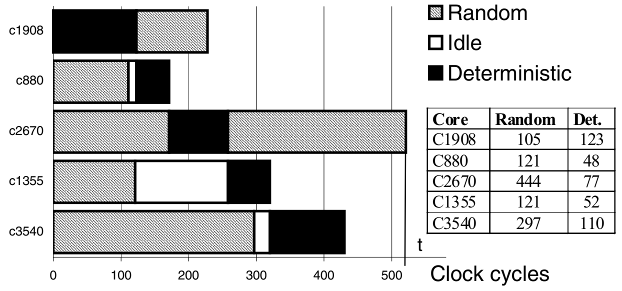


**Figure 8.3.** An example of a core-based system, with independent BIST resources

The deterministic tests, on the other hand, can only be carried out for one core at a time, which means only one test access bus at the system level is needed. An example of a multi-core system with such a test architecture is given in Figure 8.3.

This example system consists of five cores (different ISCAS benchmarks). Using the hybrid BIST optimization methodology, we can find the optimal combination between pseudorandom and deterministic test patterns for every individual core (Figure 8.4). Considering the test architecture assumed, only one deterministic test set can be applied at any given time, while any number of pseudorandom test sessions can take place in parallel. To enforce the assumption that only one deterministic test can be applied at a time, a simple *ad hoc* scheduling method can be used. The result of this schedule defines the starting moments for every deterministic test session, the memory requirements, and the total test length *t* for the whole system. This situation is illustrated in Figure 8.4.



| Core | Random | Det. |
|---|---|---|
| C1908 | 105 | 123 |
| C880 | 121 | 48 |
| C2670 | 444 | 77 |
| C1355 | 121 | 52 |
| C3540 | 297 | 110 |

**Figure 8.4.** *Ad hoc* test schedule for hybrid BIST of the core-based system example

As can be seen from Figure 8.4, the solution where every individual core has the best possible combination between pseudorandom and deterministic patterns usually does not lead to the best system-level test solution. In the example, we have illustrated three potential problems:

- The total test length of the system is determined by the single longest individual test set, while other tests may be substantially shorter.
- The resulting deterministic test sets do not take into account the memory requirements, imposed by the size of the on-chip memory or the external test equipment.
- The proposed test schedule may introduce idle periods, due to the test conflicts between the deterministic tests of different cores.

There are several possibilities for improvement. For example the *ad hoc* solution can easily be improved by using a better scheduling strategy. This, however, does not necessarily lead to a significantly better solution, as the ratio between pseudorandom and deterministic test patterns for every individual core is not changed. Therefore, we have to explore different combinations between pseudorandom and deterministic test patterns for every individual core in order to find a solution where the total test length of the system is minimized and memory constraints are satisfied. In the following sections we will define this problem more precisely, and describe a fast iterative algorithm for calculating the optimal combination between different test sets for the whole system.

### Basic Definitions and Problem Formulation

Let us assume that a system *S* consists of *n* cores $C_1$, $C_2$,..., $C_n$. For every core $C_k \in S$ a complete sequence of deterministic test patterns $\mathrm{TD}^F{}_k$ and a complete sequence of pseudorandom test patterns $\mathrm{TP}^F{}_k$ can be generated. It is assumed that both test sets can obtain by itself maximum achievable fault coverage $F_{\mathrm{max}}$.

***Definition 8.1:*** A *hybrid BIST* set $\mathrm{TH}_k = \{\mathrm{TP}_k, \mathrm{TD}_k\}$ for a core $C_k$ is a sequence of tests constructed from the subsets of pseudorandom test sequence $\mathrm{TP}_k \subseteq \mathrm{TP}^F{}_k$ and a deterministic test sequence $\mathrm{TD}_k \subseteq \mathrm{TD}^F{}_k$. The sequences $\mathrm{TP}_k$ and $\mathrm{TD}_k$ complement each other to achieve the maximum achievable fault coverage.

***Definition 8.2:*** A pattern in a pseudorandom test sequence is called *efficient* if it detects at least one new fault that is not detected by the previous test patterns in the sequence. The ordered sequence of efficient patterns form an *efficient pseudorandom test sequence* $\mathrm{TPE}_k = (P_1, P_2,\ldots, P_n) \subseteq \mathrm{TP}_k$. Each efficient pattern $P_j \in \mathrm{TPE}_k$ is characterized by the length of the pseudorandom test sequence $\mathrm{TP}_k$, from the start to the efficient pattern $P_j$, including $P_j$. Efficient pseudorandom test sequence $\mathrm{TPE}_k$, which includes all efficient patterns of $\mathrm{TP}^F{}_k$, is called a *full efficient pseudorandom test sequence* and denoted by $\mathrm{TPE}^F{}_k$.

***Definition 8.3:*** The cost of a hybrid test set $\mathrm{TH}_k$ for a core $C_k$ is determined by the total length of its pseudorandom and deterministic test sequences, which can be characterized by their costs, $\mathrm{COST}_{\mathrm{P},k}$ and $\mathrm{COST}_{\mathrm{D},k}$ respectively:

$$\mathrm{COST}_{\mathrm{T},k} = \mathrm{COST}_{\mathrm{P},k} + \mathrm{COST}_{\mathrm{D},k} = \alpha\left|\mathrm{TP}_k\right| + \beta_k\left|\mathrm{TD}_k\right|$$

and by the cost of recourses needed for storing the deterministic test sequence $\mathrm{TD}_k$ in the memory:

$$\mathrm{COST}_{\mathrm{M},k} = \gamma_k\left|\mathrm{TD}_k\right|$$

The parameters $\alpha$ and $\beta_k$ can be introduced by the designer to align the application times of different test sequences. For example, when a test-per-clock BIST scheme is used, a new test pattern can be generated and applied in each clock cycle and in this case $\alpha = 1$. The parameter $\beta_k$ for a particular core $C_k$ is equal to the total number of clock cycles needed for applying a deterministic test pattern from the memory. In a special case, when deterministic test patterns are applied by external test equipment, application of deterministic test patterns may be up to one order of magnitude slower than applying BIST patterns. The coefficient $\gamma_k$ is used to map the number of test patterns in the deterministic test sequence $\mathrm{TD}_k$ into the memory recourses, measured in bits.

***Definition 8.4:*** When assuming the test architecture described above, a hybrid test set $\mathrm{TH} = \{\mathrm{TH}_1, \mathrm{TH}_2, \ldots, \mathrm{TH}_n\}$ for a system $S = \{C_1, C_2, \ldots, C_n\}$ consists of hybrid tests $\mathrm{TH}_k$ for each individual core $C_k$, where pseudorandom components of the TH can be scheduled in parallel, whereas the deterministic components of TH must be scheduled in sequence due to the shared test resources.

***Definition 8.5:*** $J = (j_1, j_2,\ldots, j_n)$ is called the *characteristic vector* of a hybrid test set $\mathrm{TH} = \{\mathrm{TH}_1, \mathrm{TH}_2, \ldots, \mathrm{TH}_n\}$, where $j_k = |\mathrm{TPE}_k|$ is the length of the efficient pseudorandom test sequence $\mathrm{TPE}_k \subseteq \mathrm{TP}_k \subseteq \mathrm{TH}_k$.

According to Definition 8.2, for each $j_k$ there corresponds a pseudorandom subsequence $\mathrm{TP}_k(j_k) \subseteq \mathrm{TP}^F{}_k$, and, according to Definition 8.1, any pseudorandom test sequence $\mathrm{TP}_k(j_k)$ should be complemented with a deterministic test sequence, denoted with $\mathrm{TD}_k(j_k)$, that is generated in order to achieve the maximum achievable fault coverage. Based on this we can conclude that the characteristic vector $J$ determines entirely the structure of the hybrid test set $\mathrm{TH}_k$ for all cores $C_k \in S$.

***Definition 8.6:*** The test length of a hybrid test TH = {TH$_1$, TH$_2$, …, TH$_n$} for a system $S = \{C_1, C_2, \ldots, C_n\}$ is given by

$$\mathrm{COST_T} = \max\{\max_k(\alpha|\mathrm{TP}_k| + \beta_k|\mathrm{TD}_k|), \sum_k \beta_k|\mathrm{TD}_k|\}$$

The total cost of resources needed for storing the patterns from all deterministic test sequences TD$_k$ in the memory is given by

$$\mathrm{COST_M} = \sum_k \gamma_k|\mathrm{TD}_k|$$

***Definition 8.7:*** Let us introduce a generic cost function $\mathrm{COST}_{\mathrm{M},k} = f_k(\mathrm{COST}_{\mathrm{T},k})$ for every core $C_k \in S$, and an integrated generic cost function $\mathrm{COST_M} = f_k(\mathrm{COST_T})$ for the whole system $S$.

The functions $\mathrm{COST}_{\mathrm{M},k} = f_k(\mathrm{COST}_{\mathrm{T},k})$ will be created in the following way. Let us have a hybrid BIST set $\mathrm{TH}_k(j) = \{\mathrm{TP}_k(j), \mathrm{TD}_k(j)\}$ for a core $C_k$ with $j$ efficient patterns in the pseudorandom test sequence. By calculating the costs $\mathrm{COST}_{\mathrm{T},k}$ and $\mathrm{COST}_{\mathrm{M},k}$ for all possible hybrid test set structures $\mathrm{TH}_k(j)$, *i.e.*, for all values $j = 1, 2, \ldots, |\mathrm{TPE}^F{}_k|$, we can create the cost functions $\mathrm{COST}_{\mathrm{T},k} = f_{\mathrm{T},k}(j)$, and $\mathrm{COST}_{\mathrm{M},k} = f_{\mathrm{M},k}(j)$. By taking the inverse function $j = f'_{\mathrm{T},k}(\mathrm{COST}_{\mathrm{T},k})$, and inserting it into the $f_{\mathrm{M},k}(j)$, we get the generic cost function $\mathrm{COST}_{\mathrm{M},k} = f_{\mathrm{M},k}(f'_{\mathrm{T},k}(\mathrm{COST}_{\mathrm{T},k})) = f_k(\mathrm{COST}_{\mathrm{T},k})$, where the memory costs are directly related to the lengths of all possible hybrid test solutions.

The integrated generic cost function $\mathrm{COST_M}=f(\mathrm{COST_T})$ for the whole system is the sum of all cost functions $\mathrm{COST}_{\mathrm{M},k} = f_k(\mathrm{COST}_{\mathrm{T},k})$ of individual cores $C_k \in S$.

From the function $\mathrm{COST_M} = f(\mathrm{COST_T})$, the value of $\mathrm{COST_T}$ for every given value of $\mathrm{COST_M}$ can be found. The value of $\mathrm{COST_T}$ determines the lower bound of the length of the hybrid test set for the whole system. To find the component $j_k$ of the characteristic vector ***J***, *i.e.*, to find the structure of the hybrid test set for all cores, the equation $f_{\mathrm{T},k}(j)= \mathrm{COST_T}$ should be solved.

The objective here is to find a shortest possible (min(COST$_\mathrm{T}$)) hybrid test sequence TH$_{\mathrm{opt}}$ when the memory constraints are not violated $\mathrm{COST_M} \le \mathrm{COST_{M,LIMIT}}$.

### Hybrid Test Sequence Computation Based on Cost Estimates

By knowing the generic cost function $\mathrm{COST_M} = f(\mathrm{COST_T})$, the total test length $\mathrm{COST_T}$ at any given memory constraint $\mathrm{COST_M} \le \mathrm{COST_{M,LIMIT}}$ can be found in a straightforward way. However, the procedure to calculate the cost functions $\mathrm{COST}_{\mathrm{D},k}(j)$ and $\mathrm{COST}_{\mathrm{M},k}(j)$ is very time consuming, since it assumes that the deterministic test set TD$_k$ for each $j = 1, 2, \ldots, |\mathrm{TPE}^F{}_k|$ has to be available. This assumes that after every efficient pattern $P_j \in \mathrm{TPE}_k \subseteq \mathrm{TP}_k$, $j = 1, 2, \ldots, |\mathrm{TPE}^F{}_k|$ a set of not-yet-detected faults $F_{\mathrm{NOT},k}(j)$ should be calculated. This can be done either by repetitive use of the ATPG or by systematically analyzing and compressing the fault tables for each $j$ (see Section 8.2.1). Both procedures are accurate but time consuming and, therefore, not feasible for larger designs. To overcome the complexity explosion problem we propose an iterative algorithm, where costs $\mathrm{COST}_{\mathrm{M},k}$ and $\mathrm{COST}_{\mathrm{D},k}$ for the deterministic test sets TD$_k$ can be found based on estimates. The estimation method is based on fault coverage figures and does not require accurate calculations of the deterministic test sets for not-yet-detected faults $F_{\mathrm{NOT},k}(j)$.

In the following we will use $FD_k(i)$ and $FPE_k(i)$ to denote the fault coverage figures of the test sequences $TD_k(i)$ and $TPE_k(i)$ respectively, where $i$ is the length of the test sequence.

***Procedure 1:*** Estimation of the length of the deterministic test set $TD_k$.

1. Calculate, by fault simulation, the fault coverage functions $FD_k(i)$, $i = 1, 2, \ldots, |TD^F_k|$, and $FPE_k(i)$, $i = 1, 2, \ldots, |TPE^F_k|$. The patterns in $TD^F_k$ are ordered in such the way that each pattern put into the sequence contributes with maximum increase in fault coverage.
2. For each $i^* \le |TPE^F_k|$, find the fault coverage value $F^*$ that can be reached by a sequence of patterns $(P_1, P_2, \ldots, P_{i^*}) \subseteq TPE_k$ (see Figure 8.5).
3. By solving the equation $FD_k(i) = F^*$, find the maximum integer value $j^*$ that satisfies the condition $FD_k(j^*) \le F^*$. The value of $j^*$ is the length of the deterministic sequence $TD_k$ that can achieve the same fault coverage $F^*$.
4. Calculate the value of $|TD^E_k(i^*)| = |TD^F_k| - j^*$ which is the number of test patterns needed from the $TD^F_k$ to reach to the maximum achievable fault coverage.

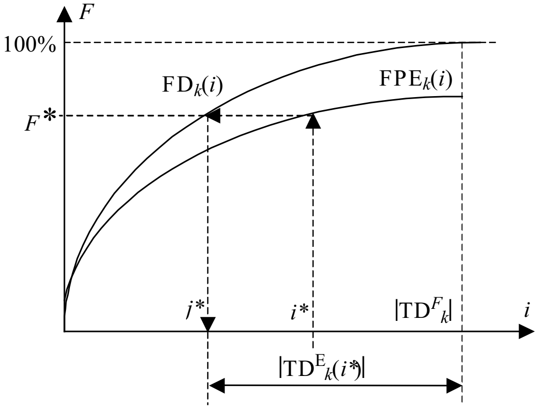


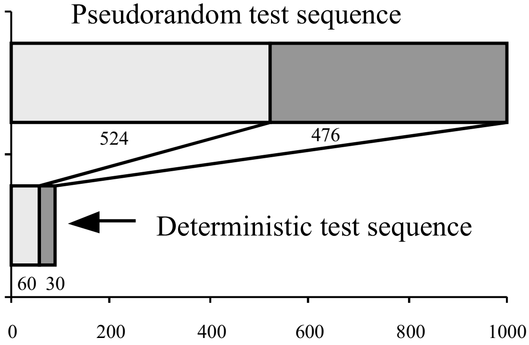
**Figure 8.5.** Estimation of the length of the deterministic test sequence

The value $|TD^E_k(i^*)|=|TD^F_k| - j^*$, calculated by Procedure 1, can be used to estimate the length of the deterministic test sequence $TD_k$ in the hybrid test set $TH_k = \{TP_k, TD_k\}$ with $i^*$ efficient test patterns in $TP_k$, ($|TPE_k|= i^*$).

By finding $|TD^E_k(j)|$ for all $j = 1, 2, \ldots, |TPE^F_k|$ we get the cost function estimate $COST^E_{D,k}(j)$. Using $COST^E_{D,k}(j)$, other cost function estimates $COST^E_{M,k}(j)$, $COST^E_{T,k}(j)$ and $COST^E_{M,k} = f_k^E(COST^E_{T,k})$ can be created according to the Definitions 8.3 and 8.7.

Finally, by adding cost estimates $COST^E_{M,k} = f_k^E(COST^E_{T,k})$ of all cores, we get the hybrid BIST cost function estimate $COST^E_M = f^E(COST^E_T)$ for the whole system.

This estimation mechanism is illustrated on Figure 8.6. It depicts fault simulation results of both pseudorandom (TP) and deterministic (TD) test sets for a given core. The length of the pseudorandom sequence has to be only so long as potentially interesting. By knowing the length of the complete deterministic test set and fault coverage figures for every individual pattern we can estimate the size of the additional deterministic test set for any length of the pseudorandom test sequence, as illustrated in the Figure 8.6. Here, we can see that for a given core 60 deterministic test cycles are needed to obtain the same fault coverage as 524 pseudorandom test cycles and it requires an additional 30 deterministic test cycles to reach 100% fault coverage. Based on this information, we assume that if we apply those 30 deterministic test cycles on top of the 524 pseudorandom cycles, we can obtain close to the maximum fault coverage. This assumption is the basis of the cost estimation procedure. Obviously, this cannot be used as a final solution; but, as we will demonstrate, it can be used as a good starting point for a test time minimization algorithm.



| \|TP\| | FC% | \|TD\| | FC% |
|---|---|---|---|
| 1 | 21.9 | 1 | 43.3 |
| 2 | 34.7 | 2 | 45.6 |
| … | | | |
| 524 | 97.5 | 60 | 97.5 |
| … | | | |
| 1000 | 98.9 | 90 | 100 |

**Figure 8.6.** Estimation of the length of the deterministic test sequence (core s1423)

### *Test Length Minimization Under Memory Constraints*

As described above, the exact calculations for finding the cost of the deterministic test set $\mathrm{COST}_{\mathrm{M},k} = f_k(\mathrm{COST}_{\mathrm{T},k})$ are very time consuming. Therefore, we will use the cost estimates, calculated by Procedure 1 in the previous section, instead. Using estimates can give us a quasi-minimal solution for the test length of the hybrid test at given memory constraints. After obtaining a quasi-minimal solution, the cost estimates can be improved and another, better, quasi-minimal solution can be calculated. This iterative procedure will be continued until we reach the final solution.

***Procedure 2:*** Test length minimization.

1. Given the memory constraint $\mathrm{COST}_{\mathrm{M,LIMIT}}$, find the estimated total test length $\mathrm{COST}^{\mathrm{E*}}_{\mathrm{T}}$ as a solution to the equation $f^{\mathrm{E}}(\mathrm{COST}^{\mathrm{E}}_{\mathrm{T}}) = \mathrm{COST}_{\mathrm{M,LIMIT}}$.
2. Based on $\mathrm{COST}^{\mathrm{E*}}_{\mathrm{T}}$, find a candidate solution $\boldsymbol{J}^* = (j^*_1, j^*_2,\ldots, j^*_n)$ where each $j^*_k$ is the maximum integer value that satisfies the equation $\mathrm{COST}^{\mathrm{E}}_{\mathrm{T},k}(j^*_k) \leq \mathrm{COST}^{\mathrm{E*}}_{\mathrm{T}}$.
3. To calculate the exact value of $\mathrm{COST}^*_{\mathrm{M}}$ for the candidate solution $\boldsymbol{J}^*$, find the set of not-yet-detected faults $F_{\mathrm{NOT},k}(j^*_k)$ and generate the corresponding deterministic test set $\mathrm{TD}^*_k$ by using an ATPG algorithm.
4. If $\mathrm{COST}^*_{\mathrm{M}} = \mathrm{COST}_{\mathrm{M,LIMIT}}$, go to the Step 9.
5. If the difference $|\mathrm{COST}^*_{\mathrm{M}} - \mathrm{COST}_{\mathrm{M,LIMIT}}|$ is bigger than that in the earlier iteration make a correction $\Delta t = \Delta t/2$, and go to Step 7.
6. Calculate a new test length $\mathrm{COST}^{\mathrm{E,N}}_{\mathrm{T}}$ from the equation $f^{\mathrm{E}}_k(\mathrm{COST}^{\mathrm{E}}_{\mathrm{T}}) = \mathrm{COST}^{*}_{\mathrm{M}}$, and find the difference $\Delta t = \mathrm{COST}^{\mathrm{E},*}_{\mathrm{T}} - \mathrm{COST}^{\mathrm{E,N}}_{\mathrm{T}}$.
7. Calculate a new cost estimate $\mathrm{COST}^{\mathrm{E},*}_{\mathrm{T}} = \mathrm{COST}^{\mathrm{E},*}_{\mathrm{T}} + \Delta t$ for the next iteration.
8. If the value of $\mathrm{COST}^{\mathrm{E},*}_{\mathrm{T}}$ is the same as in an earlier iteration, go to Step 9, otherwise go to Step 2.
9. *END:* The vector $\boldsymbol{J}^* = (j^*_1, j^*_2,\ldots, j^*_n)$ is the solution.

To illustrate the above procedure, an example of the iterative search for the shortest length of the hybrid test is given in Figures 8.7 and 8.8. Figure 8.7 represents all the basic cost curves $\mathrm{COST}^{\mathrm{E}}_{\mathrm{D},k}(j)$, $\mathrm{COST}^{\mathrm{E}}_{\mathrm{P},k}(j)$, and $\mathrm{COST}^{\mathrm{E}}_{\mathrm{T},k}(j)$, as functions of the length $j$ of $\mathrm{TPE}_k$ where $j_{\min}$ denotes the optimal solution for a single core hybrid BIST optimization problem [9].

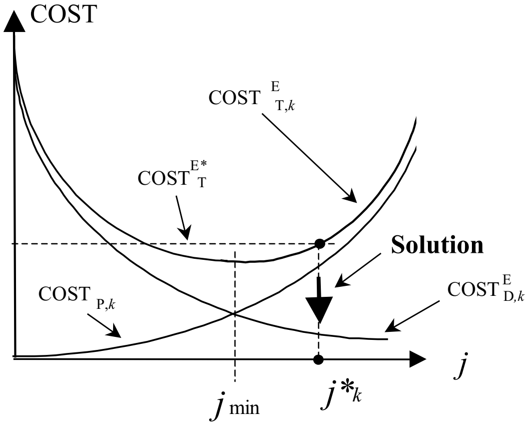


**Figure 8.7.** Cost curves for a given core $C_k$

Figure 8.8 represents the estimated generic cost function $\mathrm{COST}^{\mathrm{E}}_{\mathrm{M}} = f^{\mathrm{E}}(\mathrm{COST}^{\mathrm{E}}_{\mathrm{T}})$ for the whole system. First (Step 1), the estimated $\mathrm{COST}^{\mathrm{E*}}_{\mathrm{T}}$ for the given memory constraints is found (point 1 on Figure 8.8). Then (Step 2), based on $\mathrm{COST}^{\mathrm{E}^*}{}_{\mathrm{T}}$ the length $j^*_k$ of $\mathrm{TPE}_k$ for the core $C_k$ in Figure 8.7 is found. This procedure (Step 2) is repeated for all the cores to find the characteristic vector ***J**** of the system as the first iterative solution. After that the real memory cost $\mathrm{COST}^{\mathrm{E}^*}{}_{\mathrm{M}}$ is calculated (Step 3, point 1* in Figure 8.8). As we see in Figure 8.8, the value of $\mathrm{COST}^{\mathrm{E}^*}{}_{\mathrm{M}}$ in point 1* violates the memory constraints. The difference $\Delta t_1$ is determined by the curve of the estimated cost (Step 6). After correction, a new value of $\mathrm{COST}^{\mathrm{E}^*}{}_{\mathrm{T}}$ is found (point 2 on Figure 8.8). Based on $\mathrm{COST}^{\mathrm{E}^*}{}_{\mathrm{T}}$, a new ***J**** is found (Step 2), and a new $\mathrm{COST}^{\mathrm{E}^*}{}_{\mathrm{M}}$ is calculated (Step 3, point 2* in Figure 8.8). An additional iteration via points 3 and 3* can be followed in Figure 8.8.

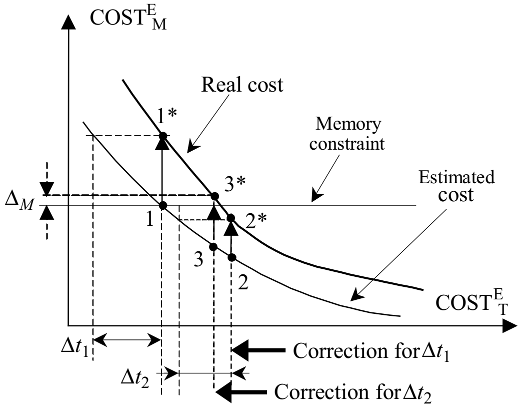
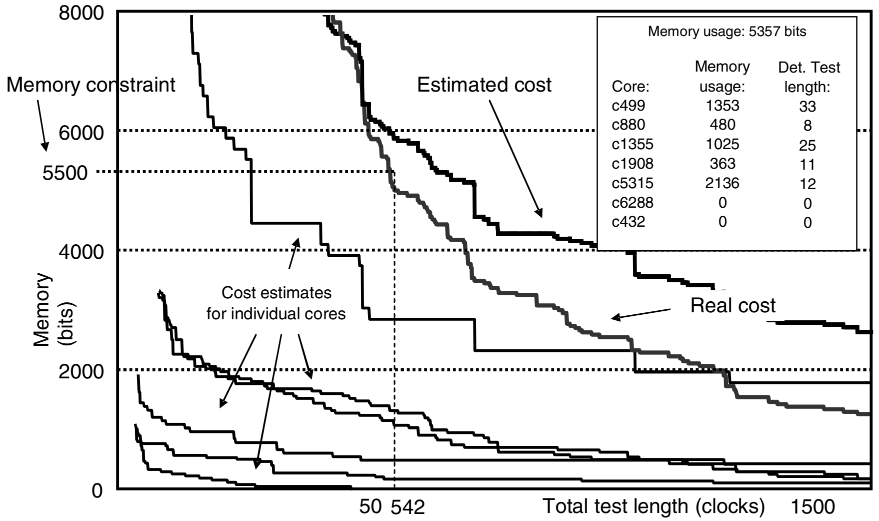


**Figure 8.8.** Minimization of the test length

It is easy to see that Procedure 2 always converges. By each iteration we get closer to the memory constraints level, and also closer to the minimal test length at given constraints. However, the solution may be only near-optimal, since we only evaluate solutions derived from estimated cost functions.



Memory usage: 5357 bits

| Core: | Memory usage: | Det. Test length: |
|---|---|---|
| c499 | 1353 | 33 |
| c880 | 480 | 8 |
| c1355 | 1025 | 25 |
| c1908 | 363 | 11 |
| c5315 | 2136 | 12 |
| c6288 | 0 | 0 |
| c432 | 0 | 0 |

**Figure 8.9.** The final test solution for the system S2 ($M_{\mathrm{LIMIT}}$ = 5500)

In Figure 8.9 we present the estimated cost curves for the individual cores and the estimated and real cost curves for one of the systems with seven cores (different ISCAS benchmarks). We also show in this picture a test solution point for this system under given memory constraint that has been found based on our algorithm. In this example we have used a memory constraint $M_{\mathrm{LIMIT}}$ = 5500 bits. The final test length for this memory constraint is 542 clock cycles, and that gives us the test schedule depicted in Figure 8.10.

This approach can easily be extended to systems with full-scan sequential cores. The main difference lies in the fact that in the case of the test-per-scan scheme the test application is done via scan chains and one test cycle takes longer than one clock cycle. This is valid for both pseudorandom and deterministic tests. As every core contains scan chains with different lengths, the analysis procedure has to follow this, and switching from one core to another has to honor the local, core-level test cycles. In the following the experimental results with systems where every individual core is equipped with a Self-Test Using MISR and a Parallel Shift Register Sequence Generator (STUMPS) [2] are presented.

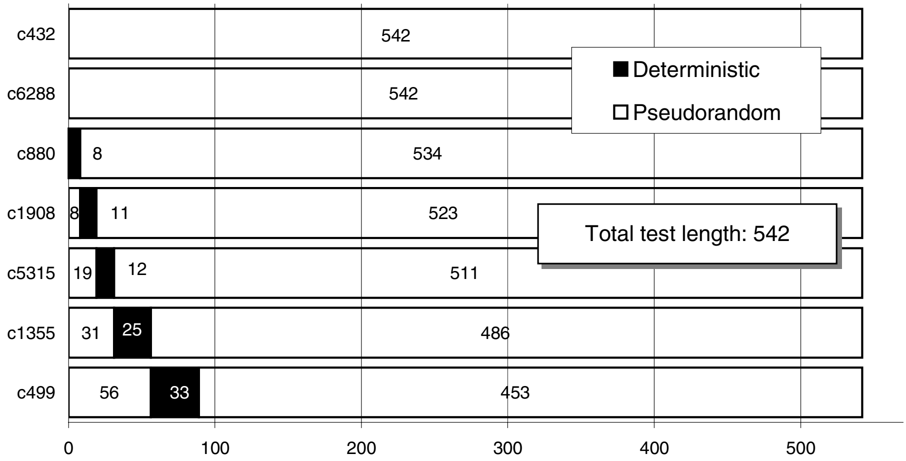


**Figure 8.10.** Test schedule for the system S2 ($M_{\text{LIMIT}} = 5500$)

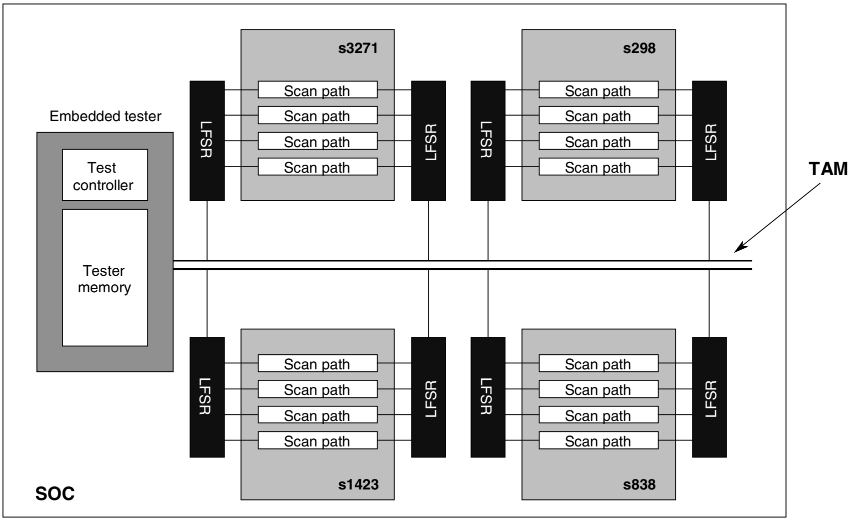


**Figure 8.11.** A core-based system example with the proposed test architecture

While every core has its own STUMPS architecture, at the system level we assume the same architecture as described earlier: every core's BIST logic is capable of producing a set of independent pseudorandom test patterns, *i.e.* the pseudorandom test sets for all the cores can be carried out simultaneously. The deterministic tests, on the other hand, can only be carried out for one core at a time, which means only one test access bus at the system level is needed. An example of a multi-core system with such a test architecture is given in Figure 8.11.

Experiments have been performed with several systems composed from different ISCAS'89 benchmarks as cores. All cores have been redesigned to include full scan path (one or several). The STUMPS architecture was simulated in software and for deterministic test pattern generation a commercial ATPG tool was used. The results are presented in Table 8.4.

**Table 8.4.** Experimental results

| | | | Exhaustive approach | | Optimized approach | |
|---|---|---|---|---|---|---|
| System name | Number of cores | Memory constraint (bits) | Total test length (clocks) | CPU time (s.) | Total test length (clocks) | CPU time (s.) |
| J | 6 | 25 000 | 5750 | 57540 | 5775 | 270 |
| | | 22 000 | 7100 | | 7150 | 216 |
| | | 19 000 | 9050 | | 9050 | 335 |
| K | 6 | 22 000 | 5225 | 53640 | 5275 | 168 |
| | | 17 000 | 7075 | | 7075 | 150 |
| | | 13 000 | 9475 | | 9475 | 427 |
| L | 6 | 15 000 | 3564 | 58740 | 3570 | 164 |
| | | 13 500 | 4848 | | 4863 | 294 |
| | | 12 200 | 9350 | | 9350 | 464 |

In Table 8.4 we compare our approach, where the test length is found based on estimates, with an exact approach where deterministic test sets have been found by a brute force method (repetitive use of a TPG) for every possible switching point between pseudorandom and deterministic test patterns. As can be seen from the results, our approach can give significant speed up (several orders of magnitude), while retaining very high accuracy.

### 8.4.2 Built-in Self-test Time Minimization for Systems with Test Pattern Broadcasting

In the previous section we analyzed systems where every core has its own dedicated BIST logic that is capable of producing a set of independent pseudorandom test patterns. This approach can be extended for multi-core systems where both combinational cores and sequential cores with full scan are used. This, however, may lead to high area overhead and may require redesign of the cores, as not all cores may be equipped with self-test structures. Therefore, we have recently proposed a novel self-test architecture that is based on test pattern broadcasting [19]. In this approach, only a single pseudorandom TPG is used and all test patterns are broadcast simultaneously for all cores in the system. These patterns will be complemented with dedicated deterministic patterns for every individual core, if needed. Those deterministic test vectors are generated during the development process and are stored in the system.

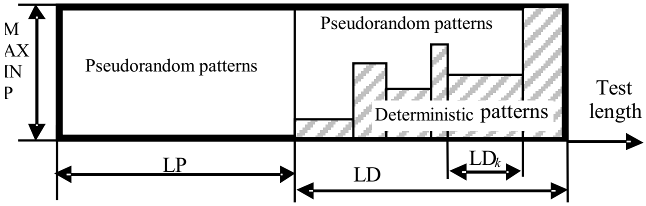


**Figure 8.12.** Hybrid test set example

Let us assume a system $S$, consisting of $n$ cores $C_1$, $C_2$, …, $C_n$, that are all connected to a bus. A hybrid test set TH = {TP, TD} for parallel testing of all the cores $C_k \in S$ is composed from the pseudorandom test set TP and deterministic test set TD. The deterministic test sequence is assembled, in general, from deterministic test sequences for each individual core TD = {TD$_1$, TD$_2$, …, TD$_n$}. Testing of all cores is carried out in parallel, *i.e.*, all pseudorandom patterns as well as each deterministic test sequence TD$_k$ are applied to all cores in the system. The deterministic test sequence TD$_k$ is a deterministic test sequence generated only by analyzing the core $C_k \in S$. For the rest of the cores $C_j \in S$, $1 \leq j \neq k \leq n$, this sequence can be considered as a pseudorandom sequence. This form of parallel testing is usually referred to as test pattern broadcasting [13]. The width of the hybrid test sequence TH is equal to MAXINP=max{INP$_k$}, $k$=1, 2, …, $n$, where INP$_k$ is the number of inputs of the core $C_k$. For each deterministic test set TD$_k$, where INP$_k$ < MAXINP, the not-specified bits will be completed with pseudorandom data, so that the resulting test set TD$_k$* can be applied in parallel to the other cores in the system as well. An example of such a hybrid test set is presented in Figure 8.12.

In Figure 8.12, we denote with LP the length of the pseudorandom test set, with LD the length of the entire deterministic test set, and with LD$_k$ the length of the deterministic test set of core $C_k$. Since some of the cores may be 100% testable by using only the pseudorandom test sequence and the deterministic test sequences of other cores, the deterministic test sequence TD$_k$ for such a core $C_k$ is not needed and LD$_k$ = 0.

The memory size for storing the deterministic part of the hybrid test set can be found from the following formula:

$$\mathrm{COST_M} = \sum_{k=1}^{n} (\mathrm{LD}_k * \mathrm{INP}_k) \tag{8.2}$$

The main problem is to minimize the total length

$$\mathrm{LH} = \mathrm{LP} + \sum_{k=1}^{n} \mathrm{LD}_k \tag{8.3}$$

of the hybrid test set TH = {TP, TD} under given memory constraint COST$_\mathrm{M}$ ≤ COST$_\mathrm{M,LIMIT}$.

The problem of minimizing the hybrid BIST length at the given memory constraints for parallel multi-core testing is extremely complex. The main reasons of this complexity are the following:

- The deterministic test patterns of one core are used as pseudorandom test patterns for all other cores; unfortunately, there will be $n*n$ relationships for $n$ cores to analyze to find the optimal combination; on the other hand, the deterministic test sets are not readily available (see Algorithm 5, later in this section) and calculated only during the analysis process.
- For a single core an optimal combination of pseudorandom and deterministic patterns can be found by rather straightforward algorithms; but, as the optimal time moment for switching from pseudorandom to deterministic testing will be different for different cores, the existing methods cannot be used and the parallel testing case is considerably more complex.
- For each core, the best initial state of the LFSR can be found experimentally; but, to find the best LFSR for testing all cores in parallel is a very complex and time-consuming task.

To cope with the high complexity of the problem we propose the following algorithm:

***Algorithm 3:***

1. Find the best initial state for the LFSR that can generate the shortest common pseudorandom sequence $TP_{INITIAL}$, sufficient for testing simultaneously all the cores with maximum achievable fault coverage. For practical reasons the $TP_{INITIAL}$ might be unacceptably long and, therefore, an adequately long $TP'_{INITIAL}$ should be chosen and complemented with an initial deterministic test set $TD_{INITIAL}$ in order to achieve maximum achievable fault coverage and to satisfy the basic requirements for the test length.
2. Based on our estimation methodology (Section 8.3.1) find the length $LD_k^E$ of the estimated deterministic test set $TD_k^E$ and calculate the first iteration of the optimized test structure $TH^E = \{TP*, TD^E\}$, so that the memory constraints are satisfied. TP* denotes here a shortened pseudorandom sequence, found during the calculations.
3. Find the real total test length LH and the real memory cost $COST_M$ of the hybrid test sequence $TH = \{TP*, TD\}$ for the selected pseudorandom sequence TP*.
4. If the memory constraints are not satisfied, *i.e.*, $COST_M > COST_{M,LIMIT}$, improve the estimation, choose a new pseudorandom sequence TP*, and repeat step 3.
5. If the memory limit has not been reached, *i.e.*, $COST_M < COST_{M,LIMIT}$, reduce the length of TH by moving efficient pseudorandom patterns [19] from the pseudorandom test set to the deterministic test set. A pattern in a pseudorandom test sequence is called *efficient* if it detects at least one new fault for at least one core that is not detected by previous test patterns in the sequence.

### Hybrid Test Sequence Computation Based on Cost Estimates

In this section we explain the first two steps of Algorithm 3. It is assumed that we have found the best configuration (polynomial and initial state) for the parallel pseudorandom TPG. Let us call this an initial pseudorandom test sequence $\mathrm{TP}_{\mathrm{INITIAL}}$.

#### Estimation of the Cost of the Deterministic Test

By knowing the structure of the hybrid test set TH, the total hybrid test length LH at any given memory constraint $\mathrm{COST}_{\mathrm{M}} \leq \mathrm{COST}_{\mathrm{M,LIMIT}}$ could be found in a straightforward way. However, calculation of the exact hybrid test structure is a costly procedure, since it assumes that for each possible length of TP the deterministic test sets $\mathrm{TD}_k$ for each core should be calculated and compressed while following the broadcasting idea. This can be done either by repetitive use of the ATPG or by systematically analyzing and compressing the fault tables. Both procedures are accurate but time consuming and, therefore, not feasible for larger designs.

To overcome the high complexity of the problem we propose an iterative algorithm, where the values of $\mathrm{LD}_k$ and $\mathrm{COST}_{\mathrm{M},k}$ for the deterministic test sets $\mathrm{TD}_k$ can be found based on estimates. The estimation method, which is an extension of the method proposed for sequential hybrid BIST (see Section 8.3.1), is based on the fault coverage figures of $\mathrm{TD}_k$ only, and does not require accurate calculations of the deterministic test sets for not-yet-detected faults.

The estimation method requires the following: a complete deterministic test set for every individual core, $\mathrm{TD}_k$, together with fault simulation results of every individual test vector $\mathrm{FD}_k$ and fault simulation results of the pseudorandom sequence $\mathrm{TP}_{\mathrm{INITIAL}}$ for every individual core, $\mathrm{FP}_k$. Let us denote with $\mathrm{TP}_{\mathrm{INITIAL}}(i)$ a pseudorandom sequence with length $i$.

The length of the deterministic test sequence $\mathrm{LD}_k(i)$ and the corresponding memory cost $\mathrm{COST}_{\mathrm{M},k}(i)$ for any length of the pseudorandom test sequence $i \leq \mathrm{LP}$ can be estimated for every individual core with the following algorithm:

***Algorithm 4:***

For each $i$ =1, 2, …, $\mathrm{LD}_k$:

1. Find fault coverage value $F(i)$ that can be reached by a sequence of pseudorandom patterns $\mathrm{TP}_{\mathrm{INITIAL}}(i)$.
2. Find the highest integer value $j$, where $\mathrm{FD}_k(j) \leq F(i)$. The value of $j$ is the required length of the deterministic sequence $\mathrm{TD}_k$ to achieve fault coverage $F(i)$.
3. Calculate the estimated length of the deterministic test subsequence $\mathrm{TD}^{\mathrm{E}}_k(i)$ as $\mathrm{LD}^{\mathrm{E}}_k(i) = \mathrm{LD}_k - j$. This is the estimated number of deterministic test patterns needed to complement the pseudorandom sequence $\mathrm{TP}_{\mathrm{INITIAL}}(i)$, so that 100% fault coverage can be achieved.

This algorithm enables us to estimate the memory requirements of the hybrid BIST solution for any length of the pseudorandom sequence for every individual core, and by adding the memory requirements of all individual cores $C_k \in S$ also for the entire system. In a similar manner, the length of the pseudorandom sequence LP for any memory constraint can be estimated, and this defines uniquely the structure of the entire hybrid test set.

### Computation and Minimization of the Hybrid Test Sequence

The memory cost estimation function helps us to find the length LP* of the pseudorandom test sequence TP* for the estimated hybrid test sequence $TH^E=\{TP^*; TD^E\}$. The real length LH of the estimated hybrid test sequence $TH^E$ can be found with the following algorithm.

***Algorithm 5:***

1. Simulate the pseudorandom sequence TP* for each core $C_k \in S$ and find a set of not-detected faults $F_{NOT,k.}$ Generate the corresponding deterministic test set $TD'_k$ by using any ATPG tool. As a result, a preliminary real hybrid test set will be generated: TH = {TP*; TD'}.
2. Order the deterministic test set TD' = $(TD'_1, TD'_2, \ldots, TD'_n)$ in such the way that for each $i < n$, $INP_i \leq INP_{i+1}$.
3. Perform the analysis of the test pattern broadcasting impact for $i = 2, 3, \ldots n$:
   - calculate a set of not-detected faults $F_{NOT,i}$ for the test sequence $(TP^*; TD'_1, TD'_2, \ldots, TD'_{i-1})$;
   - compress the test patterns in $TD'_i$ with respect to $F_{NOT,k}$ by using any test compacting tool.

As a result of Algorithm 5, the real hybrid test sequence TH = {TP*; TD} = $\{TP^*; TD_1, TD_2, \ldots, TD_n\}$ will be generated. The length of the resulting sequence $LH \leq LH^E$ as deterministic test patterns of one core, while broadcast to the other cores, may detect some additional faults. In general, $LD_k \leq LD^E_k$ for every $k = 2, 3, \ldots, n$.

The length of the deterministic test sequence, generated with Algorithm 5, can be considered as a near-optimal solution for the given test access mechanism, for all the cores. Ordering of the deterministic test sets according to step 2 in Algorithm 5 has the following result: the larger the number of inputs of core $C_k$ the more patterns will broadcast to $C_k$ from other cores, and hence the chances to reduce its own deterministic test set $TD_k$ are bigger and larger amounts of memory can be reduced.

After finding the real deterministic test sequence according to Algorithm 5, the following three situations may occur:

1. If $COST_M > COST_{M,LIMIT}$ a new iteration of the cost estimation should be carried out. The initial estimation of the pseudorandom test sequence length LP should be updated, and a new cost calculation, based on Algorithm 5, should be performed.
2. If $COST_M = COST_{M,LIMIT}$ the best possible solution for the given pseudorandom sequence TP* is found. TH = $\{TP^*; TD_1, TD_2, \ldots, TD_n\}$.
3. If $COST_M < COST_{M,LIMIT}$ the test length minimization process should be continued by moving efficient test patterns from the pseudorandom test set to the deterministic sequence.

In the following, possible steps for further improvement are described in detail.

### Iterative Procedure for Cost Estimation

Let us suppose that our first estimated solution, based on pseudorandom test sequence TP, with length LP, produces a test structure with total memory requirement “Real COSTM” higher than accepted (see Figure 8.13). A correction of the estimated solution should be made LPNEW = LP + ΔLP and a new solution “New real $COST_M$” should be calculated based on Algorithm 5. Those iterations should be repeated until the memory constraint $COST_M \le COST_{M,LIMIT}$ is satisfied.

It should be mentioned that Algorithm 5 is the most expensive procedure of the whole approach, due to repetitive use of ATPG and test compaction tools. Therefore, we cannot start with an arbitrary initial solution, and an accurate estimation procedure minimizes the number of iterations considerably.

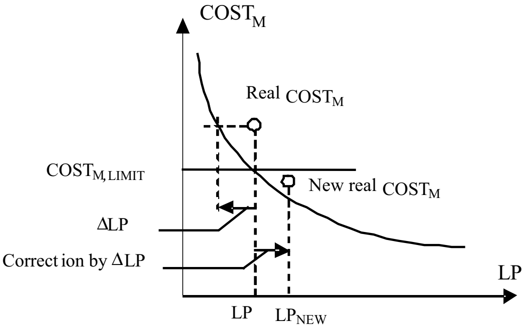


**Figure 8.13.** Iterative cost estimation

### Total Test Length Reduction by Reducing the Pseudorandom Test Sequence

Suppose that the real cost of the solution found is below the memory constraint $COST_M < COST_{M,LIMIT}$. There are two alternatives for further reduction of the test length:

1. Additional iterations by using Algorithm 5 to move the solution as close to the memory limit $COST_{M,LIMIT}$ as possible. As mentioned earlier, Algorithm 5 is an expensive procedure and, therefore, recommended to be used as little as possible.
2. It is possible to minimize the length of the hybrid test sequence TH by shortening the pseudorandom sequence, *i.e.*, by moving step-by-step efficient patterns from the beginning of TP to TD and by removing all other patterns between the efficient ones from TP, until the memory constraint $COST_M \le COST_{M,LIMIT}$ gets violated. This procedure is based on the algorithm used in [19] for straightforward optimization of the parallel hybrid BIST. As a result, the final hybrid test sequence is created: $TH_F = \{TP_F; TD_F\} = \{TP_F; TD_1, TD_2, \ldots, TD_n, \Delta TD\}$ where ΔTD is a set of efficient test patterns moved from TP to TD. This will lead to the situation where the length of the pseudorandom sequence has been

reduced by ΔLP and the length of the deterministic test sequence has been increased by ΔLD. The total length $LH_F$ of the resulting hybrid test $TH_F = \{TP_F; TD_F\}$ is shorter, $LH_F < LH$, because in general ΔLD << ΔLP (not every pattern in the pseudorandom test set is efficient).

The final hybrid BIST test structure $TH_F = \{TP_F; TD_F\}$ with the total length $LH_F$ is represented in Figure 8.14.

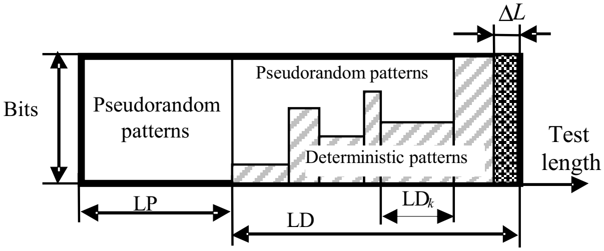


**Figure 8.14.** Final hybrid test structure

The accuracy of the solution (proximity of the total length $LH_F$ to the global minimum $LH_{MIN}$) for the given initial pseudorandom sequence $TP_{INITIAL}$ can be estimated by the length of ΔLD, assuming that the deterministic test set was optimally compacted. Since efficient patterns, moved from TP to TD, were not taken into account during the compaction procedure for TD’ (Algorithm 5), the new deterministic test sequence $TD_F = \{TD_1, TD_2,\ldots, TD_n, \Delta TD\}$ is not optimal and should be compacted as well. However, since TD’ was compacted optimally, the upper bound of the gain in test length cannot be higher than ΔLD. Hence, the difference between the exact minimum $LH_{MIN}$ and the current solution $LH_F$ for the given pseudorandom sequence $TP_{INITIAL}$ cannot be higher than $LH_F - LH_{MIN} = \Delta LH$.

We have performed experiments with three systems composed from different ISCAS benchmarks as cores. In Table 8.5 the results are compared with the straightforward approach, *i.g.*, in fact, the fifth step of the Algorithm 3 [19]. The length of the pseudorandom test sequence, deterministic test sequence and the hybrid test sequence, together with required CPU time, are compared. The first component in the deterministic test column represents the result of Algorithm 5, and the second component represents the last improvement, when efficient patterns were moved from the pseudorandom part to the deterministic part. As can be seen, the proposed approach gives a noteworthy reduction of the test length over the straightforward approach, while the analysis time is approximately the same.

**Table 8.5.** Comparison with straightforward approach

| | | Straightforward approach | | | |
|---|---|---|---|---|---|
| System | Memory constraint (bits) | PR length (clocks) | DET length (clocks) | Total length (clocks) | CPU time (s) |
| S1 (6 cores) | 10000 | 232 | 105 | 337 | 187.64 |
| | 10000 | 250 | 133 | 383 | |
| S2 (7 cores) | 5000 | 598 | 71 | 669 | 718.49 |
| | 3000 | 819 | 48 | 867 | |
| S3 (5 cores) | 10000 | 465 | 161 | 626 | 221.48 |
| | | | Our approach | | |
| S1 (6 cores) | 10000 | 145 | 58 + 49 = 107 | 252 | 289,73 |
| | 10000 | 163 | 110 + 14 = 124 | 287 | 1093,5 |
| S2 (7 cores) | 5000 | 469 | 51 + 18 = 69 | 538 | 1124,4 |
| | 3000 | 783 | 23 + 19 = 42 | 825 | 1109,4 |
| S3 (5 cores) | 10000 | 262 | 130 + 10 = 140 | 402 | 334,28 |

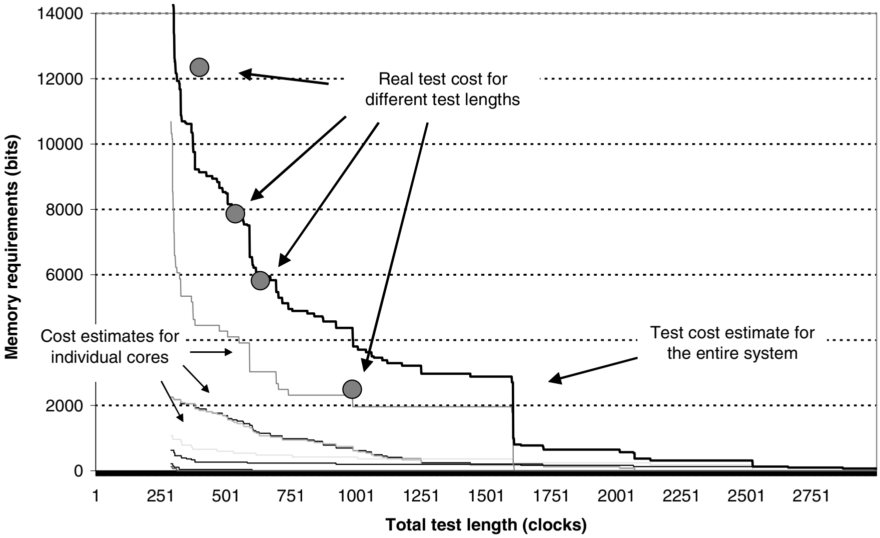


**Figure 8.15.** Comparison of estimated and real test costs

Figure 8.15 depicts the estimated memory cost as a function of the total test length for different cores in system S2 together with the estimated total memory cost. For comparison, the real cost values for four different test lengths are shown as well. As can be seen, the accuracy of the estimation procedure is rather good.

## 8.5 Conclusions

This chapter presented the hybrid BIST optimization problem as an example of system-level DfT methods. The main objective of deploying the hybrid BIST technique is to improve fault coverage by mixing pseudorandom vectors with deterministic ones. We have mainly discussed the issues related to test cost minimization and, in particular, test time minimization with several different implementations of the hybrid BIST architecture. The optimization algorithms have been presented together with experimental results to demonstrate their efficiency.

## References

[1] Agrawal VD, Kime CR, Saluja KK (1993) A tutorial on built-in self-test. IEEE Design and Test of Computers, (March): 69-77.

[2] Bardell PH, McAnney WH, Savir J (1987) Built-in test for VLSI pseudorandom techniques. John Wiley and Sons.

[3] Brglez F, Fujiwara H (1985) A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran. In: Proc. IEEE Int. Symp. on Circuits and Systems, 663-698.

[4] Chatterjee M, Pradhan DK (1995) A novel pattern generator for near-perfect fault-coverage. In: Proc. IEEE VLSI Test Symposium, 417-425.

[5] Glover F (1986) Future paths for integer programming and links to artificial intelligence. Computers & Ops. Res., (5): 533-549.

[6] Golomb SW (1982) Shift register sequences. Aegan Park Press, Laguna Hills.

[7] Hellebrand S, Tarnick S, Rajski J, Courtois B (1992) Generation of vector patterns through reseeding of multiple-polynomial linear feedback shift registers. In: Proc. IEEE Int. Test Conference, 120-129.

[8] Hellebrand S, Wunderlich H-J, Hertwig A (1998) Mixed-mode BIST using embedded processors. Journal of Electronic Testing: Theory and Applications, (12): 127-138.

[9] Jervan G, Peng Z, Ubar R (2000) Test cost minimization for hybrid BIST. In: Proc. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 283-291.

[10] Jervan G, Peng Z, Ubar R, Kruus H (2002) A hybrid BIST architecture and its optimization for SOC testing. In: Proc. IEEE International Symposium on Quality Electronic Design, 273-279.

[11] Jervan G, Eles P, Peng Z, Ubar R, Jenihhin M (2003) Test time minimization for hybrid BIST of core-based systems. In: Proc. 12th IEEE Asian Test Symposium, 318-323.

[12] Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimization by simulated annealing. Science, 220(4598): 671-680.

[13] Lee K-J, Chen J-J, Huang C-H (1999) Broadcasting test patterns to multiple circuits. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 18(12): 1793-1802.

[14] Sugihara M, Date H, Yasuura H (2000) Analysis and minimization of test time in a combined BIST and external test approach. In: Proc. IEEE Design, Automation & Test In Europe Conference, 134-140.

[15] Touba NA, McCluskey EJ (1995) Synthesis of mapping logic for generating transformed pseudo-random patterns for BIST. In: Proc. IEEE Int. Test Conference, 674-682.

[16] Tallinn Technical University (1999) Turbo Tester Reference Manual. Version 3.99.03, http://www.pld.ttu.ee/tt

[17] Ubar R, Jervan G, Peng Z, Orasson E, Raidma R (2001) Fast test cost calculation for hybrid BIST in digital systems. In: Proc. Euromicro Symposium on Digital Systems Design, 318-325.

[18] Ubar R, Kruus H, Jervan G, Peng Z (2001) Using Tabu search method for optimizing the cost of hybrid BIST. In: Proc. 16th Conference on Design of Circuits and Integrated Systems, 445-450.

[19] Ubar R, Jenihhin M, Jervan G, Peng Z (2004) Hybrid BIST optimization for core-based systems with test pattern broadcasting. In: Proc. IEEE Int. Workshop on Electronic Design, Test and Applications, 3-8.

[20] Yarmolik VN, Kachan IV (1993) Self-checking VLSI design. Elsevier Science Ltd

[21] Zacharia N, Rajski J, Tyzer J (1995) Decompression of test data using variable-length seed LFSRs. IN: Proc. IEEE 13th VLSI Test Symposium, 426-433.

# 9 System-level Dependability Analysis

A. Bobbio, D. Codetta Raiteri, M. De Pierro, and G. Franceschinis

Università del Piemonte Orientale, Dipartimento di Informatica, Alessandria, Italy
Università di Torino, Dipartimento di Informatica, Torino, Italy

## 9.1 Abstract

The focus of this work is on the dependability analysis of safety or mission-critical systems; in particular, we concentrate on the control subsystem, which is made up of several *components*. We assume that the components, which may be designed with the support of hardware–software codesign tools, are characterized by dependability (*e.g.* failure rate) parameters, which may derive from simulators of the components while they are under development, or as a result of testing (possibly combined with fault injection techniques). By using combinatorial and state-space-based techniques it is possible to derive the reliability of the whole system as a function of the system configuration and of the component parameters values, and to identify the criticality of a given component or subset of components. The analysis is performed by applying *Fault Tree Analysis* (FTA) techniques enhanced with recently introduced features that allow one to remove the components' independence assumptions imposed by classical FTA, and to include the possibility of component as well as subsystem repair.

## 9.2 Introduction

This work is about dependability analysis of safety or mission-critical systems; in particular, we concentrate on the control subsystem, which is made up of several *components*, both hardware and software, which may be distributed and communicate through some kind of interconnection network. The components, which may be designed with the support of hardware–software codesign tools, are characterized by dependability (*e.g.* failure rate) and performance parameters (*e.g.* average response time), which may derive from simulators of the components while they are under development, or as a result of testing (possibly combined with fault injection techniques) after a prototype is available. By using combinatorial and state-space-based models it is possible to derive dependability measures or performance measures of the whole system, as a function of the system configuration and of the component parameters values.

The components may be subject to faults, which can be internal faults due to the component characteristics (perhaps due to some undetected problem in the test phase), or externally induced faults, due to interference of the environment in which the control system operates (*e.g.*, bit flips in memory components due to electromagnetic interference). Component faults may occur independently, or may exhibit some form of correlation. Moreover, if faults can be detected, appropriate recovery actions can be taken (*e.g.*, the faulty component may be replaced).

The dependability analysis of such systems can be performed by means of combinational models representing the structure of the system in terms of error propagation from components to subsystems, up to the whole system, or by means of behavioral models, representing the reachable system states and the possible state transitions, that can be simulated (or, under some constraints, solved analytically). Usually, the former types of model can be solved by resorting to (rather efficient) combinatorial methods, while the latter are more expensive (depending on the number of possible states of the model). Efficient combinatorial methods, however, can be applied only in the hypothesis of independence of the components failure.

In [28] a hierarchy of models of increasing expressive power has been presented: among the combinatorial models, Fault Tree (FT) with repeated events is the more powerful; however, FT Analysis (FTA) techniques can be applied only under some restrictive assumptions, namely the failure of each component cannot depend on the state of other components; moreover, once a component is down it cannot be repaired. When these assumptions are not realistic for the system under study, state-space-based methods can be applied (in [28] Continuous Time Markov Chains (CTMCs), stochastic Petri nets and Markov reward models are proposed to this purpose): the problem in this case is due to the state-space size, which grows exponentially with respect to the number of components, so that their analysis might have very high computational costs, or even be unfeasible.

Several research studies have recently been published that try to combine the advantages of combinatorial and state-space-based analysis methods; the common underlying idea is to isolate minimal subsystems that must be treated by resorting to state-space-based methods, and then combine them in an FT-like structure, exploiting combinatorial analysis techniques at the overall system level. In order to avoid exposing the system designer to several different dependability model languages, the description of the subsystems including dependencies should preferably be expressed by using a language similar to the original FT formalism; for this reason, several extensions of the FT formalism have been proposed, which can be translated by means of automatic tools into state-space models. Also, the hybrid combinatorial–state-space analysis techniques must be applied transparently by isolating and solving the submodels requiring state-space-based analysis, and then combining the results by means of classical FT combinatorial techniques [5], [9], [13], [23], [27], [29].

In this chapter we show the evolution of FT models by means of intuitive examples, from the classical version to increasingly more powerful extensions. First, the FT language extensions are motivated and described on practical examples.

Then the types of measure that can be derived from such models are defined; finally, the analysis results for the illustrative examples are shown and discussed.

## 9.3 Introduction to Fault Trees

The FT [15],[37] is a widespread stochastic model for the reliability analysis of complex systems because it provides an intuitive representation of the system failure mode, it is easy to manipulate and it is currently supported by several software tools for its analysis. An FT models how combinations of failure events relative to the components of the system can cause the failure of subsystems or of the whole system. Let us introduce the FT formalism by means of the example depicted in Figure 9.1.

The FT is a bipartite *Direct Acyclic Graph* (DAG) whose nodes can belong to one of two categories: events and gates. Events concern the failure of components, subsystems or of the whole system, and they are in general graphically represented as a box; we can consider an event as a Boolean variable: it is initially *false* and it becomes *true* after the failure occurrence.

The events represented as a box with an attached circle are called *Basic Events* (BEs) and model the failure of the elementary components of the system; the occurrence time of such events is ruled by a probability distribution, typically a negative exponential, whose parameter $\lambda$ is called *failure rate* and is equal to the inverse of the mean life time of the component. The BEs are the leaves[1] of the FT. The events represented simply by a box are non-terminal nodes and represent the failure of subsystems; we call them “Internal Events” (IEs) and their occurrence is not ruled by a probability distribution as in the case of BEs, but they are the output of a gate node; gates are the other category of nodes an FT can contain, and they are connected by means of arcs to several input events and to a unique output event; the effect of a gate is the propagation of the failure to its output event if a particular combination of its input events occurs; in the standard version of the FT model, three combinational gates corresponding to the AND, OR and “*K* out of *N*” (*K*:*N*) Boolean functions, are defined.

The Boolean value of the output event of a gate is determined by applying the function associated with the gate to the values of the gate input events: considering an IE that is the output of an AND gate, it will occur (it is set to *true*) if all the input events of such a gate have occurred (all of them have a *true* value); if an IE is the output of an OR gate, it will occur if at least one of the gate input events has occurred; in the case of the *K*:*N* gate, the output event occurs if at least *K* of the *N* input events have occurred. The occurrence of an IE is immediate, as soon as the particular combination of input events required by the gate is verified. In the FT graphical notation, arcs have no orientation, but we can say that they respect a

[1] We call “leaves of the FT” the BE abusing terminology, even if the FT is actually a DAG, not a tree.

logic circuit orientation: from the input events to the gate, and from the gate to the output event.

Finally, we have a unique event, represented as a black box, called a *Top Event* (TE), modeling the failure of the whole system; the TE must be the output of a gate (moreover, it must be the unique node which is not input to any gate) and cannot be the input of any gate; we can consider it as the root of the FT.

### 9.3.1 Fault Tree Example

Let us consider a simple real case of a system whose failure mode we want to model using the FT formalism: the state variables storage for the control system of a primary station in a power distribution mesh. The storage system is composed by a set of memories to store the variables, and by a voter unit; the values stored inside a single memory may not be correctly updated (or may be corrupted due to electromagnetic interference), so when the value of a variable is requested, the voter forwards a request to every memory and the correct value of the variable is obtained by voting on the several values returned by the several memories or it will answer with a value different than the others, so that the voter shall detect the failure and exclude the faulty memory from later requests. In order to increase the dependability of the system, a set of hot spare memories have been added to the system; in other words, the spare memories are updated each time the variable is modified, but initially they do not belong to the voting set. At the same time, spare memories may fail as the main ones if one of the memories does not respond to the voter request or returns a value in disagreement with the majority.

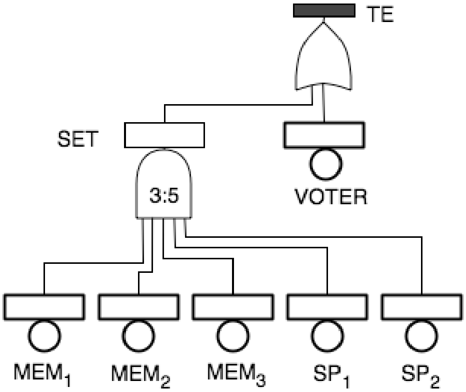


**Figure 9.1.** The FT for the storage system with hot spare memories

Figure 9.1 shows the system FT assuming that the voting group must be composed of three memories and two spare memories are available; the TE represents the failure of the whole system and is the output of an OR gate with the two input events SET and VOTER, which are an IE and a BE respectively; this means that the system fails if the voting cannot be performed because there are not enough working memories to compose the voting set or the voter unit has failed. The event SET is the output of a *K*:*N* gate (in this case *K*=3 and *N*=5) whose input events are the BEs called $\text{MEM}_1$, $\text{MEM}_2$, $\text{MEM}_3$, $\text{SP}_1$ and $\text{SP}_2$ to represent the failures of the three main memories and of the two spare memories. Since the voting set is assumed to be composed of three elements and at most two main memories can be replaced in the voting set by the spare memories, the voting cannot be performed if at least three failures occur among the main and the spare memories; in other words, the system is tolerant of one or two memory failures, but it is not tolerant of three or more failures.

### 9.3.2 Modeling Dependencies in the Failure Mode Using Dynamic Gates

The FT formalism has a very intuitive notation, but it suffers from the inability to model dependencies among failure events or component states; this is due to the assumption in the standard version of this model that BEs are statistically independent. In order to overcome this limitation, some new gates called *dynamic gates* [17],[26],[27] were introduced with the aim of modeling several kinds of dependency among the events, leading to the Dynamic FT (DFT) formalism; let us give the definition of such gates:

- *Functional Dependency* (FDEP) *gate* – given as input events a trigger event *T* and a set of dependent events D1, ..., D*m*, the dependent events are forced to occur when the trigger event occurs; the output of the gate corresponds to the state of *T*. Note that D1, …, D*m* can be the input events of other gates.
- *Priority And* (PAND) *gate* – given X1, ..., X*n* as input events and *Y* as output event, *Y* fails if all X1, ..., X*n* have occurred and only in a specified order.
- *Sequence Enforcing* (SEQ) *gate* - given X1, ..., X*n* as input events and *Y* as output event, X1, ..., X*n* are forced to occur in a specified order; *Y* corresponds to the state of X*n*.
- *Warm Spare* (WSP) *gate* – this gate models the presence of a set of warm spare components able to replace a main component when it fails; warm spares differ from hot spares by the fact that they can be in three states instead of two: dormant (or stand-by), working, failed; a spare is initially dormant and it turns to the working state if it has to replace the main component or another spare; at the same time, a spare may fail both in the dormant and in the working state; the spare failure rate changes depending on its current state: if the failure rate of the spare is $\lambda$ in the working state, $\alpha\lambda$ is its failure rate in the dormant state, with $0<\alpha<1$; $\alpha$ is called the *dormancy factor* and its aim is to express the fact that spares have a reduced failure probability during the dormancy period. The input events of this gate are the events corresponding to the failure of the main component and the events corresponding to the failure of the spares; the output event occurs if the main component fails and there are no available spares to replace it. Figure 9.2 shows the dependency of a spare on the main component.

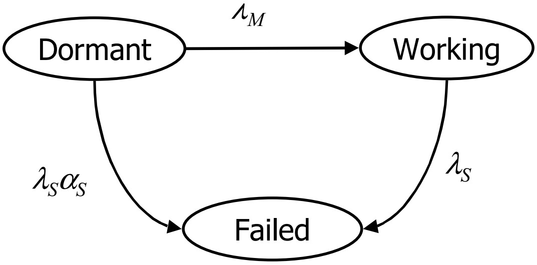


**Figure 9.2.** State-space representation of the dependency of spare *S* on main component *M*

Considering our example, we can apply the WSP gate to the memories with the resulting DFT shown in Figure 9.3: the IE named SET is now the output of an OR gate whose input events are $MEM_1$, $MEM_2$ and $MEM_3$; all of them are the output of a WSP gate having as first input $MAIN_1$, $MAIN_2$ and $MAIN_3$ respectively, while all the WSP gates are connected to $SP_1$ and $SP_2$; $MEM_1$, $MEM_2$ and $MEM_3$ represent the failure of the first, the second and the third main memory respectively, while $SP_1$ and $SP_2$ represent the spare memories; such a DFT represents the system with two warm spare memories that can replace any of the main ones; the event $MEM_1$ occurs when the first main memory has failed and there are no available spares to replace it; a spare is not available if it is already replacing another main component or it has failed[2]; $MEM_2$ and $MEM_3$ have the same meaning as $MEM_1$, but with regard to the second and the third main memory respectively; the event SET still represents the fact that the number of working memories is insufficient for the voting to take place, but now SET occurs when a main memory cannot be replaced; the failure of the whole system still occurs when the voting set elements are not enough or when the voter fails.

### 9.3.3 Giving a Compact Representation of Symmetric Systems Through Parameterization

We can give a more compact representation of the failure mode of a system containing symmetries or redundancies using the *Parametric FT* (PFT) [3],[4] formalism; in the PFT, subtrees with the same structure, the same failure rates corresponding to the BEs, and connected to the same gate, are folded in a unique parameterized subtree. The PFT formalism can be combined with the dynamic gates generating the *Dynamic PFT* (DPFT) formalism [5].

[2] The failure of a warm spare component may occur while it is dormant or while it is working, as explained in the formal definition of the WSP gate.

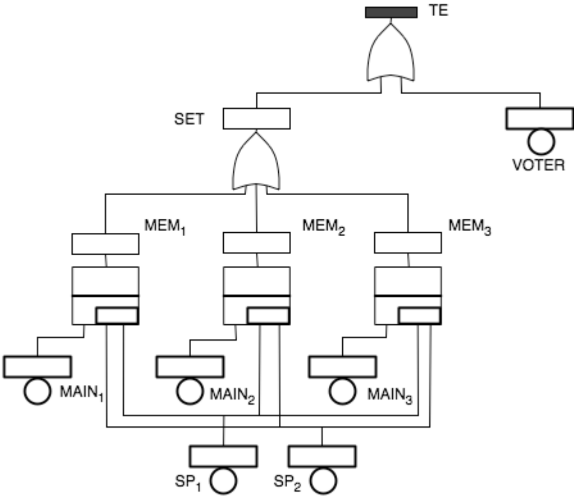


**Figure 9.3.** The DFT for the system with warm spares

Considering the version of our example with warm spares (Figure 9.3), the corresponding DPFT is shown in Figure 9.4a: assuming that all the main memories have the same failure rate and that all spare memories have the same failure rate and the same dormancy factor, the subtrees having as root $MEM_1$, $MEM_2$ and $MEM_3$ have been folded in the parameterized subtree whose root is the *Replicator Event* (RE) labeled as MEM(*i*); an RE is indicated as a dotted box and indicates the root of a parameterized subtree; one or more parameters are associated with the RE with the purpose of defining the number of identical distinct subtrees that are represented compactly; in the case of MEM(*i*), the parameter *i* has a variation range from 1 to 3 in order to express that three subtrees have been folded; the same parameter may also be associated with the events below the RE in order to indicate that a distinct copy of them is present in each folded subtree; in our case, we have MAIN(*i*).

In the PFT formalism, if an event does not have the same parameter(s) as the RE at the root of the parameterized subtree, it means that such an event belongs to all the folded subtrees; a parameterized subtree may contain other (basic) REs in order to express that each folded subtree may contain other folded subtrees. In our case, we have the *Basic RE* (BRE) called SP(*j*) with *j* equal to 1 or 2; such a BRE (graphically represented as a dotted box with an attached circle) provides a compact notation for SP1 and SP2; SP(*j*) belongs to the parameterized subtree having MEM(*i*) as root, to express compactly that any of the main memories can be replaced by any of the spare ones. In our case, the reduction in the dimensions of the model is quite visible by comparing the DFT in Figure 9.3 and the DPFT in Figure 9.4a representing the failure mode of the same system; moreover, using a DPFT, we can arbitrarily change the number of the main and spare memories without modifying the DPFT structure, just by changing the range of the parameters.

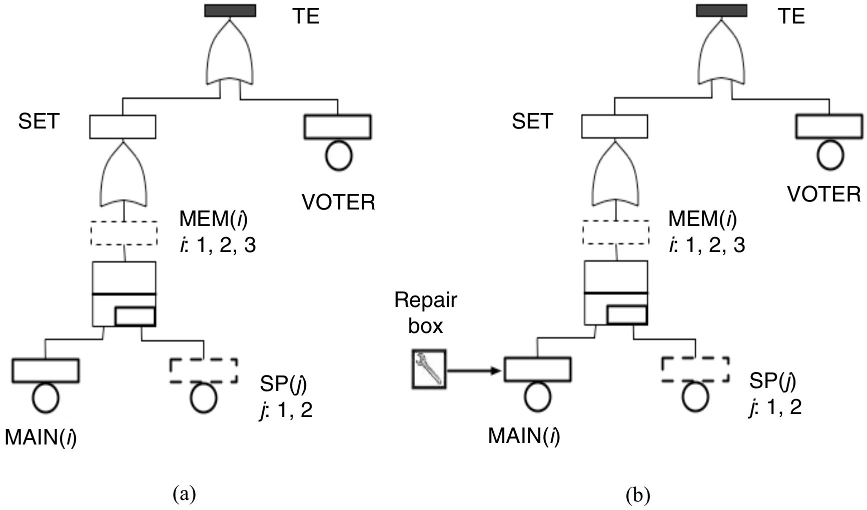


**Figure 9.4.** (a) DPFT and (b) RDPFT for the system with warm spares

### 9.3.4 Modeling the Repair Process Through the Repair Box

The DPFT formalism is not able to model a recovery or repair process; we have to extend it introducing a new type of node called a *Repair Box* (RB) [5],[13]. A repair box models the presence in the system of a repair process in order to turn a failed component or subsystem back to the working state. An RB is connected by means of an arc to the event corresponding to the failure event of the component or of the subsystem we want to be repairable; the repair time is a random variable ruled by a probability distribution function; we will use an exponential distribution whose parameter $\mu$ is called *repair rate* and is equal to the inverse of the mean time necessary to repair the component. The model we obtain introducing the RBs is called a *Repairable DPFT* (RDPFT) [5].

Let us provide now an example of the use of an RB in our case study, in its version with warm spares; Figure 9.4b is the corresponding RDPFT: MAIN(*i*) is connected to an RB, graphically represented as a wrench inside a square, to represent that this failure is repairable; since the repair box is connected to a parameterized event, we can imagine having a single RB for each main memory. Considering the events as Boolean variables, we can say that the effect of the repair box action consists of resetting the value of MAIN(*i*) to *false* and this may lead to resetting to *false* even the values of those events depending on the repaired component; in this case, they are SET and TE. So, the repair of a single component may influence even the state of a subsystem or of the whole system.

In this case, since some warm spares are present, the RB influences their state too; a main memory repair process begins when the main memory fails; during the repair period the main memory may be replaced by a spare one which turns from the dormant state to the working state; when the repair ends, the spare that is currently replacing the main memory can turn back again into the dormant state and can be used again into replace a main memory. So, the state of a spare depends both on the failure and the repair of the main memory.

Several repair policies can be associated with an RB; in our case, we assume that the repair process is enabled as soon as the failure of the component occurs and its duration is ruled by the repair rate, with no limits regarding the number of components that must be simultaneously repaired.

The introduction of the RBs in our modeling formalism allows us to represent both the failure and the repair mode of the system; an RB can also be connected to an IE with the purpose of modeling the repair of a subsystem; in this case, the RB action is enabled when the IE occurs. The RB may affect either the whole subsystem or we can specify which components of the subsystem are repairable by connecting the corresponding BEs to the RB by means of arcs. In general, the RB has to be connected to the event enabling it (the subsystem failure) and to the BEs corresponding to the failures of the elementary components we want to be repairable. When the repair process involves several components, a repair rate for each of them must be specified or we can suppose to have a general repair rate for the subsystem repair.

Other features which may be specified for the repair process are the maximal number of components the RB can repair at the same time and the time that the RB needs to detect the failure and consequently to begin its action.

## 9.4 Reliability Analysis

In this section the FTA techniques for the (un)reliability analysis are discussed, and in the next section the application of such methods will be applied to the examples introduced above.

### 9.4.1 Qualitative Analysis

FTA provides a set of techniques enabling one to derive both qualitative results and quantitative measures useful for the assessment of mission-critical or safety-critical systems. Qualitative analysis supplies information about functional and logic properties of the system failure modes. The *Minimal Cut-Set* (MCS) [33] analysis makes available to the engineer the information about the minimal sets of basic component failures leading to a system failure; MCSs represent sets of necessary and sufficient component failures able to cause the top event. The *path-set* analysis is dual to MCS analysis and provides the sets of components that when all are working cause the system to work. Section 9.5.1 shows the MCS analysis results for the FT depicted in Figure 9.1.

The number of basic events in an MCS is called the order of the MCS. The order is a significant qualitative parameter, since it highlights failure sets of events that might be more critical for the system. In fact, an MCS of order 1 means that the failure of a single basic component is sufficient to determine the TE, indicating no fault tolerance with respect to that component. In an MCS of order 2, two simultaneous failures of basic components are needed. Many FTA tools catalog the MCS in increasing order, so that the list starts with those that are potentially most critical.

### 9.4.2 Quantitative Analysis

FT quantitative analysis presents the engineer with measures of system (un)reliability or (un)availability. The reliability of either a component or the system is defined as the probability that the component or the system is working properly at a given time. Several reliability measures can be computed on the FT; these are the top-event probability Pr{TE}, the occurrence probability of each MCS and the system Mean-Time To Failure (MTTF). Finally, the component importance factors, discussed in the next section, which give quantitative information on the criticality of each component, are also important and efficiently derivable from the FT.

Quantitative analysis is performed by providing quantitative information about the basic component unreliability or unavailability, expressed as a failure probability (*i.e.*, the probability of the component being down). From this information the whole system unreliability or unavailability can be derived according to the FT structure. Often, the failure probability of components is not directly expressed in the FT; instead, a time-to-failure distribution is provided, from which a failure probability at time $t$ can be derived. Typically, the distribution is a negative exponential with parameter $\lambda_i$, so that the probability that component $i$ is down at time $t$ can be computed through the following formula:

$$q_i(t) = 1 - \mathrm{e}^{-\lambda_i t} \tag{9.1}$$

If the basic components can be repaired then they can move from the up (working) to the down state and back, so that it is possible to compute the unavailability of the component in a given time interval (ratio between the down time and the global interval duration). The unavailability of the whole system can then be computed as a function of the unavailability of its components by applying the same procedure used for computing the system unreliability.

The most common measure used to assess the system safety is its unreliability in time. The system unreliability, denoted by $Q$, is a function of the basic component unreliability $q_i(t)$:

$$Q(\boldsymbol{q}(t)) \tag{9.2}$$

where $\boldsymbol{q}(t)$ is the vector of basic component failure probabilities. In an FT such an indicator corresponds to the top-event occurrence probability, Pr{TE}. When the basic components are stochastically independent, such as in an FT, Pr{TE} may be computed by resorting to combinatorial formulas. In fact, for a given time instant *t* and fixed basic component failure occurrence probabilities ***q***(*t*), the TE probability can be derived. This can be done by exploiting the results of the MCS analysis by using the following inclusion–exclusion expansion, where $C_i$ is the MCS:

$$\begin{aligned}\Pr\{\mathrm{TE}\} = &\sum_{i=1}^{n} \Pr\{C_i\} - \sum_{\forall i,\forall j} \Pr\{C_i \wedge C_j\} \\ &+ \sum_{\forall i,\forall j,\forall k} \Pr\{C_i \wedge C_j \wedge C_k\} + \ldots + (-1)^{n+1} \Pr\{C_1 \wedge \ldots \wedge C_n\}\end{aligned} \tag{9.3}$$

For complex systems the above formula may be prohibitive to derive due to the huge amount of calculation to be performed; as a result, most FTA tools compute an approximation based on the kinetic tree theory [41]. Recently, the computation of both qualitative results and quantitative measures has improved notably by the introduction of Binary Decision Diagrams (BDDs) by Bryant [8]. BDDs allow one to encode the Boolean function characterizing the TE in a very compact way. The BDD representation had an enormous practical impact on the computational efficiency of the FTA algorithms allowing one to derive the exact value of the indices of reliability even for a large system, without resorting to an approximate solution. However, the nature of the worst-case complexity does not change. In [7],[32],[35] innovative BDD-based algorithms showing improvement in qualitative and quantitative FTA of safety-critical industrial systems are provided.

### 9.4.3 Importance Measures

Among the risk-assessment and safety-analysis objectives, the classification of a system's components according to their criticality is very important. So, in addition to the reliability measure of a system or its own subcomponent, it is central to assess the role a component takes on with regard to the system reliability. This analysis is significant to the engineer both during the design phase and successively in quantifying the risk-importance of the various system components. Importance analysis allows us to assess which component of the system is most critical to its un-reliability, so as to find out the more cost-effective solution to improve reliability. Moreover, it permits us to evaluate how much the results depend on the accuracy of the input parameters. To this purpose, several indices, commonly called the *importance factors*, have been proposed. Importance factors are time-dependent measures and may be divided into two groups: measures calculated at one point in time, such as those discussed later, and measures whose values are obtained averaging on a time period [30].

Among the various importance factors introduced in the context of importance analysis using an FT, the most popular is due to Birnbaum [2] and is defined as the partial derivative of the system unreliability with respect to the unreliability of the component addressed. This measure is also known as the Marginal Impact

Factor (MIF). The importance of component *i* to system unreliability is defined by Birnbaum as

$$G_i(t) = \frac{\partial Q(q(t))}{\partial q_i(t)} \tag{9.4}$$

It is possibile to show that for a standard FT $G_i(t)$ is equal to:

$$G_i(t) = Q(t)[1,i] - Q(t)[0,i] \tag{9.5}$$

where $Q(t)[1,i]$ and $q(t)[0,i]$ are the system unreliability given that basic component *i* is respectively working and not working.

Others measures of component importance have been proposed in the literature and they may be found in textbooks [24],[25]. Table 9.1 shows some of them.

**Table 9.1.** Importance factors

| Criticality measure | Acronym | Definition |
|---|---|---|
| Critical Importance Factor | CIF$_i(t)$ | $G_i(t) \cdot \Pr\{i\} / \Pr\{\mathrm{TE}\}$ |
| Diagnostic Impact Factor | DIF$_i(t)$ | $\Pr\{i \mid \mathrm{TE}\} = \Pr\{\mathrm{TE} \cdot i\} / \Pr\{\mathrm{TE}\}$ |
| Risk Achievement Worth | RAW$_i(t)$ | $\Pr\{\mathrm{TE} \mid i\} / \Pr\{\mathrm{TE}\}$ |
| Risk Reduction Worth | RRW$_i(t)$ | $\Pr\{\mathrm{TE}\} / \Pr\{\mathrm{TE} \mid \bar{i}\}$ |

The Birnbaum measure for component *i* does not depend on the component reliability. Nevertheless, it is useful to evaluate the criticality that an improvement in the component *i* reliability may play in the system reliability. CIF extends the Birnbaum index to take into account such factor. DIF is also known as the Vesley–Fussel Importance factor [42] and it measures the fraction of the system unreliability involving the situations in which component *i* has failed. RAW for a given component *i* measures the increase in system failure probability, and when calculated for different values of $q_i$ it is a meter of the importance of maintaining the current level of reliability for the component *i*. The RRW of a component is the decrease in risk if the component is assumed to be perfectly reliable. It is expressed in terms of the ratio of the risk level to the risk with the component guaranteed to succeed.

Several studies [20],[21],[22],[32],[36] presented efficient algorithms to derive exactly the importance factors described above using techniques based on BDD representations. Many FTA tools exploit these techniques.

In the above discussion, only single-component importance factor measures have been introduced, but in principle it may also be relevant to compute importance factors for sets of components (*e.g.*, for the MCS). In [6],[31] Bayesian network are used to derive the posterior failure probability of a given subset of events, *i.e.*, the probability that when the TE is observed, then that subset of events has occurred.

## 9.5 Qualitative and Quantitative Analysis of the Examples

In this section, the qualitative and quantitative analysis is performed for each configuration of our system; first, we detect the MCSs of the system, then we calculate the unreliability and the importance measures for each configuration of the system.

Several tools have been used to obtain the results reported in this section; the unreliability analysis of the standard FT has been performed using SHARPE [34] and ASTRA [16]; SHARPE allows us to compute exactly the system MTTF on the FT and the whole system unreliability at several times; ASTRA returns an upper bound for the system unreliability and is useful to calculate exactly the components and MCSs' unreliability on the FT; the unreliability analysis of DPFT and RDPFT has been performed using a specific solver [14] based on modularization [1] and interacting with other tools such as DrawNET [42] (its graphical interface), GreatSPN [12] (for the state-space analysis, based on SWN) and SHARPE (for the combinatorial analysis). The unreliability analysis of the DPFT can also be performed on the corresponding DFT using the Galileo [18],[39] tool. The unreliability of the MCSs and of the dependent components concerning the DPFT and DRPFT have been calculated using state-space analysis by translating the model in SWN and solving it by means of GreatSPN.

ASTRA and Galileo allow one to calculate the importance measures of the components on the FT and DFT respectively, but ASTRA always returns approximated results because it does not calculate the system unreliability exactly; Galileo uses an approximated formula to compute the MIF when the system contains dynamic gates; so, in order to have exact importance measures always, we had to calculate them in a semi-automatic way by modifying properly the Bes' probabilities and calculating the system unreliability with SHARPE for the FT, and with GreatSPN for DPFT and RDPFT, once the model has been translated in SWN.

### 9.5.1 Minimal Cut-sets Detection

Considering the FT of Figure 9.1, the MCS analysis provided 11 MCSs: one of them has order 1, while all the others have order 3; they are as follows:

$$\begin{gathered} \mathrm{TE} = \{\ \mathrm{VOTER},\ \mathrm{MEM_1MEM_2MEM_3},\ \mathrm{MEM_1MEM_2SP_1}, \\ \mathrm{MEM_1MEM_2SP2},\ \mathrm{MEM_1MEM_3SP_1},\ \mathrm{MEM_1MEM_3SP_2}, \\ \mathrm{MEM_2MEM_3SP_1},\ \mathrm{MEM_2\ MEM_3SP_2},\ \mathrm{MEM_1SP_1SP_2},\ \mathrm{MEM_2SP_1SP_2}, \\ \mathrm{MEM_3SP_1SP_2}\ \} \end{gathered} \tag{9.6}$$

Observing the MCSs list and excluding the MCS composed only by VOTER, we can note that all the MCSs are composed of two main memories and one spare memory, or by one main memory and two spare ones; moreover, the main or spare memories composing these MCSs have a common failure rate, so the probability to occur of all the MCS having order 3, is the same; for these reasons, we can express such MCSs in a parametric form, in this way:

1. VOTER
2. $\mathrm{MEM}_1$ $\mathrm{MEM}_2$ $\mathrm{MEM}_3$
3. $\mathrm{MEM}_x$ $\mathrm{MEM}_y$ $\mathrm{SP}_z$ $\forall\, x, y, z : x, y \in \{1, 2, 3\} \wedge z \in \{1, 2\} \wedge x \neq y.$
4. $\mathrm{MEM}_i$ $\mathrm{SP}_j$ $\mathrm{SP}_k$ $\forall\, i, j, k : i \in \{1, 2, 3\} \wedge j, k \in \{1, 2\} \wedge j \neq k.$

Using this notation, a parametric MCS represents a class of ordinary MCSs that are composed of the same types of component with the same failure rate; for all the ordinary MCSs that are compactly represented by the same parametric MCS, we obtain the same values concerning the unreliability; in our case, $\mathrm{MEM}_x\mathrm{MEM}_y\mathrm{MEM}_z$ is the compact representation of six ordinary MCSs (from the second to the eighth one, in the list), while $\mathrm{MEM}_x\mathrm{SP}_y\mathrm{SP}_z$ is the parametric form of three ordinary MCSs (the last three in the list). The parametric form allows a compact and intuitive notation to express the MCSs of a system containing some kinds of symmetry or redundancy, as in our example.

Considering the DPFT of Figure 9.4a and the RDPFT of Figure 9.4b, the MCSs listed above are still valid because the combinations of components causing the TE are always the same: in the DPFT we have warm spares instead of hot spares, but the system is still tolerant to two memories failures; in the RDPFT we introduced the possibility of repairing the main memories, but the TE can still be caused by one of the MCSs listed above. Even if the MCSs of our system are always the same, their probability of occurring changes with respect to the system configuration, as we can verify by calculating the MCSs' unreliability values, which are presented in the next section.

### 9.5.2 Quantitative Analysis

Table 9.2 shows the failure rates for the components of the system, with the corresponding MTTF (the inverse of the failure rate); such rates are assigned to the corresponding BEs; all the main and spare memories have a common failure rate.

**Table 9.2**. Failure rates for the system components

| Component | Failure rate (1/h) | MTTF (h) |
|---|---|---|
| VOTER | 2.0E−05 | 50000 |
| MEM(*i*) | 1.0E−04 | 10000 |
| SP(*j*) | 1.0E−04 | 10000 |

#### *Quantitative Analysis on Fault Tree*

The MTTF of the whole system modeled as an FT (Figure 9.1) is equal to 7074 h; such measure has been calculated by means of the SHARPE tool. Table 9.3 shows the unreliability values for the system components at several times; the main and the spare memories have the same unreliability since they have the same failure rate and they are independent from each other; Table 9.4 shows the unreliability values at several times, for the MCSs and for the whole system; in such a table the unreliability of the MCS composed only by VOTER has been omitted since it is equivalent to the VOTER unreliability, while all the others MCSs have the same unreliability because all of them have the same order and are composed of components with the same unreliability at the given time.

**Table 9.3.** The unreliability values for the components of the system with hot spares

| Hours | $q$(VOTER) | $q$(MEM) = $q$(SP) |
|---|---|---|
| 2000 | 0.03921 | 0.18126 |
| 4000 | 0.07688 | 0.32967 |
| 6000 | 0.11307 | 0.45118 |
| 8000 | 0.14785 | 0.55067 |
| 10000 | 0.18126 | 0.63212 |

**Table 9.4.** Unreliability of all the MCSs with order 3, and the system

| Hours | $Q$(MCS) | $Q$(TE) |
|---|---|---|
| 2000 | 0.00595 | 0.08200 |
| 4000 | 0.03583 | 0.26565 |
| 6000 | 0.09184 | 0.47588 |
| 8000 | 0.16698 | 0.65433 |
| 10000 | 0.25258 | 0.78421 |

Importance factors are time dependent; Table 9.5 shows the MIF calculated at several times on the components; let us consider the MIF values for the voter; such a measure indicates the contribution of the voter to system failure, though the voter unreliability is generally lower than the unreliability of a memory, the voter failure is more important than a memory failure with respect to the system, especially for the lowest time values; this is due to the fact that the voter failure causes system failure directly, while a failed memory may be replaced.

Considering the highest times of Table 9.5, the MIF values of the voter and of a memory do not differ so much as for the lowest time values: the unreliability of the memories has increased enough that the probabilities of the MCSs concerning the memories are similar to the probability of the voter failure. The MIF values of the voter and of the memories are represented graphically in Figure 9.5.

**Table 9.5.** The MIF values for the components of the system with hot spares

| Hours | MIF(VOTER) | MIF(MEM) = MIF(SP) |
|---:|---:|---:|
| 2000 | 0.95545 | 0.12697 |
| 4000 | 0.79550 | 0.27049 |
| 6000 | 0.59094 | 0.32628 |
| 8000 | 0.40564 | 0.31302 |
| 10000 | 0.26356 | 0.26564 |

### *Quantitative Analysis on Dynamic Parametric Fault Tree*

Owing to the presence of dependencies among some components in the system, the unreliability values cannot always be calculated with the same procedure used for the FT; considering the DPFT of Figure 9.4a, the voter and the main memories are independent of the rest of the system, so their unreliability has not changed from the previous case; the presence of the WSP gate causes a dependency of the spare memories on the main ones, so the spares' states and the corresponding failure rates depend on the main memories' states at the current time; for this reason, the calculation of the unreliability values for the spares needs an analysis of the system behavior in the state space. In this representation, every state is a description of the working and not working components, and transitions between states are the failure and repair events.

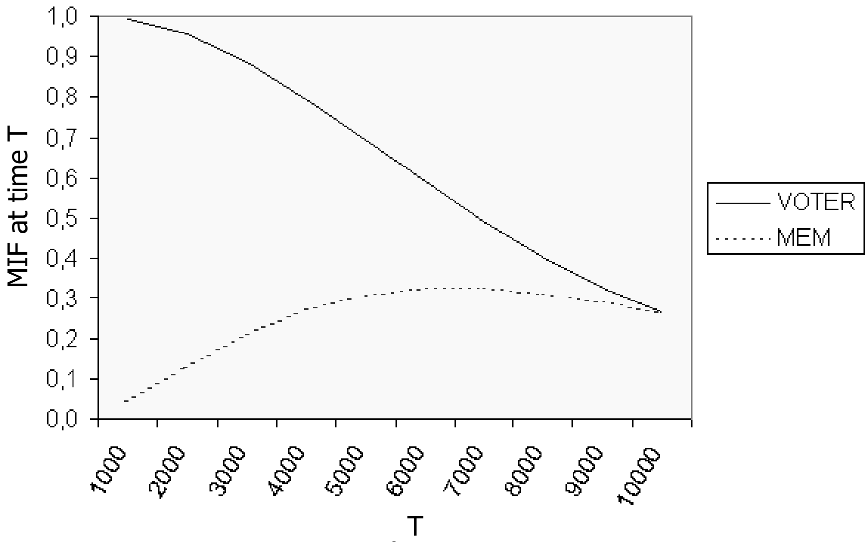


**Figure 9.5.** The MIF values for the components of the system with hot spares

State-space analysis is typically computationally expensive, so modular approaches have been developed for DFT solution; in other words, such analysis is applied only to the minimal parts of the system that need it, specifically because they contain dynamic gates. The state-space analysis can be performed by translating the *minimal independent subtree* containing dynamic gates (referred as a dynamic module [5]) in a state-space-based stochastic model such as CTMC or *Stochastic Petri Nets* (SPNs).

Generally, modules can be detected by means of a linear algorithm [19] which has been adapted to be used on DFT [1] and (R)DPFT [5]. In our example, the dynamic module consists of the subtree whose root is the event SET; such a module is parameterized, so we have a compact representation of the system behavior which can be conserved in the corresponding state-space model by translating the dynamic module in a particular form of colored stochastic Petri net called a *Stochastic Well-formed Net* (SWN) [10],[11]. An SWN allows one to generate a compact (symbolic) state space of the system, instead of the ordinary one, with a significant reduction [5] of the state-space dimensions and with a corresponding reduction of the computational costs.

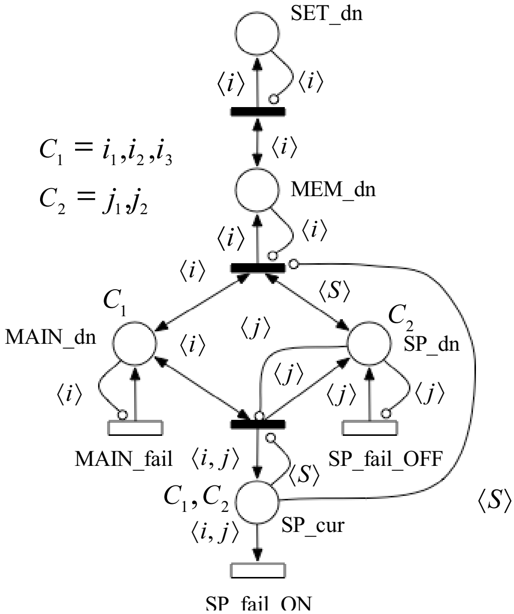


**Figure 9.6.** The SWN corresponding to the dynamic module whose root is the event SET

The SWN for the dynamic module whose root is SET is shown in Figure 9.6; we can arbitrarily calculate several indices on it, such as the unreliability value of a spare memory at a given mission time. Table 9.6 shows the unreliability values of the spare memories, of the MCSs concerning the memories, and of the whole system. The spare memories now have a lower unreliability due to their condition of warm spare (in Table 9.6 it is assumed to have a dormancy factor equal to 0.1); for the same reason, the reliability of the MCSs concerning both main and spare memories, and of the whole system, is lower than the corresponding value in the previous configuration; the probability to occur of the MCS composed of all the main memories has not changed, because the unreliability of the main memories has not changed.

On a DPFT, we can have dependency among the BEs, so the unreliability of the MCSs cannot be computed with the formula used for the FT, where BEs are assumed to be all statistically independent; on a DPFT, the unreliability of an MCS must be calculated by considering all the possible occurrence sequences of the BEs composing the MCS [40]; every sequence may have a different probability to occur, with respect to the other ones; for instance, the probability of the sequence $MEM_1 \rightarrow MEM_2 \rightarrow SP_1$, is different from the probability of $SP_1 \rightarrow MEM_1 \rightarrow MEM_2$; this is due to the fact that $SP_1$ occurs in the first sequence during its dormant period, while in the second one it occurs during the working period, with a higher failure rate.

Considering our example, the unreliability of the MCSs concerning the main and the spare memories can be calculated on the SWN, relative to the dynamic module.

**Table 9.6.** The unreliability values for the components of the system with warm spares ($\alpha$= 0.1).

| Hours | $q$(SP) | $Q$(MEM$_1$ MEM$_2$ MEM$_3$) | $Q$(MEM$_x$ MEM$_y$ SP$_z$) | $Q$(MEM$_i$ SP$_j$ SP$_k$) | $Q$(TE) |
|---|---|---|---|---|---|
| 2000 | 0.10825 | 0.00596 | 0.00471 | 0.00346 | 0.06334 |
| 4000 | 0.22491 | 0.03583 | 0.02955 | 0.02327 | 0.19613 |
| 6000 | 0.33915 | 0.09185 | 0.07842 | 0.06498 | 0.36606 |
| 8000 | 0.44414 | 0.16698 | 0.14665 | 0.12632 | 0.53191 |
| 10000 | 0.53676 | 0.25258 | 0.22703 | 0.20148 | 0.67122 |

The unreliability of the whole system modeled as a DPFT can be calculated through these steps: we have to calculate the unreliability at a given time of the dynamic module whose root is SET, in the state space, and replace it with a BE with a fixed probability to occur, equal to the unreliability of the module. Once we have replaced it, we have no dynamic gates, so we can solve the model as an FT. Such a procedure must be repeated for each time value.

Table 9.7 shows the MIF values for the components in the case of the DPFT; from such values we can see that the importance of the voter is higher than in the previous configuration; this is due to the fact the unreliability of the MCSs including spare memories has decreased because the spare memories now have a lower failure probability. So the memories have become less important as a failure cause of the system, both in the case they are main and in the case they are spare; to balance such a decrease, the voter has become more important as a failure cause than before.

**Table 9.7.** The MIF values for the components of the sytem with warm spares

| Hours | MIF(VOTER) | MIF(MAIN) | MIF(SP) |
|---|---|---|---|
| 2000 | 0.97489 | 0.11160 | 0.10635 |
| 4000 | 0.87082 | 0.26888 | 0.24403 |
| 6000 | 0.71476 | 0.36578 | 0.31602 |
| 8000 | 0.54931 | 0.39455 | 0.32444 |
| 10000 | 0.40158 | 0.37528 | 0.29371 |

### *Quantitative Analysis on Repairable Dynamic Parametric Fault Tree*

The introduction of the RB establishes for some components a dependency on the RB action; in our example the RB first influences the state of a main memory by repairing it, but also the state of the spare memories and maybe of the whole system. An analysis in the state space is necessary also in this case of using an SWN including both the failure and the repair mode of the system (we omit the corresponding figure); we have assumed that more than one main memory can be repaired at the same time, and that the repair process starts as soon as the main memory fails. Observing the results for the unreliability in Table 9.8, these show that the unreliability of a repairable main memory is the same for all the times indicated; this is due to the fact that the alternation for such a component of working and repairing periods whose durations are ruled by an exponential distribution with a constant rate leads to having a constant probability of the component being in the working or in the failed state, for times exceeding a certain limit.

The unreliability of the warm spares has decreased with respect to the previous configuration, because now a warm spare memory which is replacing a main one under repair can turn back to the dormant state when the repair ends, reducing its probability of failure.

**Table 9.8.** The unreliability values for the components of the system with repairable main memories and warm spares

| Hours | *Q*(MAIN) | *q*(SP) |
|---|---|---|
| 2000 | 0.00990 | 0.02195 |
| 4000 | 0.00990 | 0.04395 |
| 6000 | 0.00990 | 0.06552 |
| 8000 | 0.00990 | 0.08664 |
| 10000 | 0.00990 | 0.10734 |

The probabilities of the MCSs to occur are calculated using the SWN, including the repair process too; Table 9.9 shows such values whose reduction is evident and is due to the repair action directly involving the main memories and, as a secondary effect, the spare ones. The RB also influences the state of the whole system, so its unavailability can be calculated with the same method used for the DPFT; the results are shown in Table 9.9, with a significant reduction of the failure probability of the system.

**Table 9.9.** The unavailability values for the MCSs of the system with repairable main memories and warm spares

| Hours | $U(\text{MEM}_1\ \text{MEM}_2\ \text{MEM}_3)$ | $U(\text{MEM}_x\ \text{MEM}_y\ \text{SP}_z)$ | $U(\text{MEM}_i\ \text{SP}_j\ \text{SP}_k)$ | $U(\text{TE})$ |
|---|---|---|---|---|
| 2000 | 0.00005 | 0.00007 | 0.00009 | 0.03864 |
| 4000 | 0.00011 | 0.00024 | 0.00044 | 0.07641 |
| 6000 | 0.00017 | 0.00048 | 0.00119 | 0.11272 |
| 8000 | 0.00023 | 0.00081 | 0.00244 | 0.14762 |
| 10000 | 0.00028 | 0.00122 | 0.00424 | 0.18117 |

Table 9.10 shows the MIF values for the system components. The action of the RB has influenced the importance measures too; now the memories can be replaced and repaired at the same time, so a system failure due to a combination of failed memories is very unlikely, so the voter, is typically the system failure cause at any time; the MIF values of the voter compared with those of the memories, are the evident proof of this situation. The evaluation of the components, importance can be performed by resorting to other importance factors, but, due to the system structure, we would obtain similar results, so we limited our attention only on the MIF, the simplest to be calculated.

**Table 9.10.** The MIF values for the components of the system with repairable main memories and warm spares

| Hours | MIF(VOTER) | MIF(MAIN) | MIF(SP) |
|---|---|---|---|
| 2000 | 0.99993 | 0.00174 | 0.00144 |
| 4000 | 0.99985 | 0.00228 | 0.00201 |
| 6000 | 0.99975 | 0.00274 | 0.00252 |
| 8000 | 0.99961 | 0.00313 | 0.00298 |
| 10000 | 0.99946 | 0.00346 | 0.00338 |

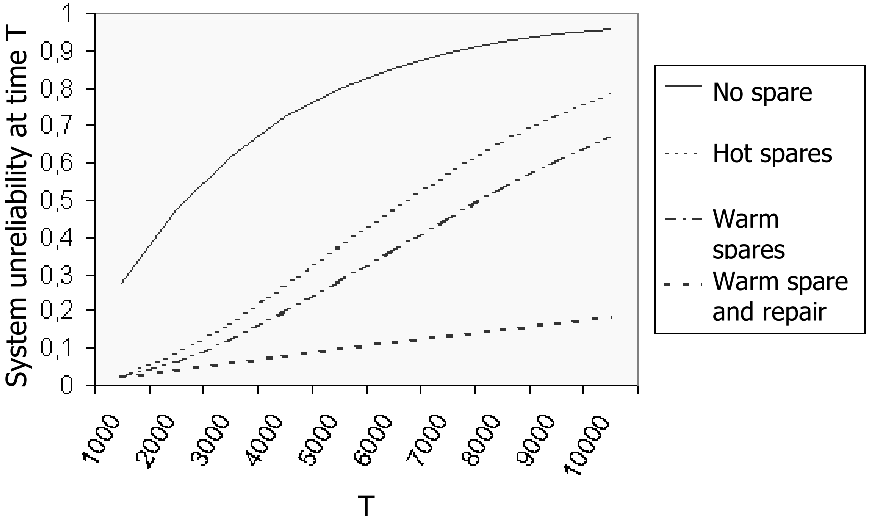


**Figure 9.7.** The unreliability values for all the system configurations

Figure 9.7 graphically compares the whole system unreliability (unavailability) for each configuration, including as upper bound the case with no spare memories, but only three main memories, which has not been considered above. The figure shows how the introduction of warm spares instead of hot spares reduces the system unreliability, while the action of an RB determines a further evident reduction of such a measure.

## 9.6 Conclusions

An FT representation of complex systems embedding several components allows one to express the failure propagation logic of the system, and to assess the system (un)reliability as a function of the components' (un)reliability. Several extensions to the standard FT language have been proposed in the literature, to enhance its expressive power and allow one to account for component dependencies or the possibility of repair. Such extensions require more expensive state-space-based analysis methods in contrast to the efficient combinatorial ones applicable to standard FTs: the analysis complexity can, however, be kept under control by applying hybrid and modular approaches, restricting state-space analysis only to those subsystems that actually need it to be solved, and then composing the results on the subsystems into a new FT representing the failure event propagation from the subsystems up to the TE, which can be solved by combinatorial methods. Such a decomposition and composition method can be done in a completely automated fashion for the classes of DPFT and DRPFT.

Several measures can be obtained from the FT models: system and MCS reliability indices, component or subsystem importance measures, MTTF of the system and of its components. In order to obtain these measures, the elements of the FT corresponding to basic components must be enriched with the quantitative parameters indicating their reliability (failure probability) or availability; these parameters can be obtained by measures performed directly on the components themselves, which could be derived through simulation at the design stage, or directly measured on the component (possibly tested in both normal mode and with fault injection).

Besides the consideration on models and tools for reliability assessment, it would be also important to have the possibility of evaluating the so-called performability measures, which are measures expressing the performance degradation of a (fault-tolerant) system as a function of the number and type of the component failure that have occurred. This aspect can be dealt with by using several techniques, typically state-space-based ones: in [38] a heterogeneous system performability and reliability study has been proposed, exploiting a hierarchical simulation environment integrating low-level simulators (the ones available in the context of synthesis of hardware–software components), able to represent in some detail the behavior of the system components, with higher level simulators capturing the structure and behavior of the system as a whole. Measures can flow from bottom to top and *vice versa*, until the method converges to a stable value. This technique, in conjunction with the extended FT analysis discussed in this chapter, can help both system and component designers to evaluate the adequacy of the design choices with respect to the reliability and performance requirements from the early design phases.

## Acknowledgments

This work has been partially funded by the MIUR FIRB project PERF (RBNE019N8N).

## References

[1] Anand A, Somani K (1998) Hierarchical analysis of fault trees with dependencies, using decomposition. In: Proc. Annual Reliability and Maintainability Symposium, 69-75

[2] Birnbaum ZW (1969) On the importance of different components and a multicomponent system. In: Korishnaiah P.R., editor, Multivariable Analysis II. Academic Press, New York

[3] Bobbio A, Franceschinis G, Gaeta R, Portinale L (2003) Parametric fault-tree for the dependability analysis of redundant systems and its high level Petri net semantics. IEEE Transactions on Software Engineering, 29: 270-287

[4] Bobbio A, Franceschinis G, Gaeta R, Portinale L (2001) Dependability assessment of an industrial programmable logic controller via parametric fault-tree and high level PN. In: Proc. 9th International Workshop on Petri Nets and Performance Models, 29-38

[5] Bobbio A, Codetta Raiteri D (2004) Parametric fault trees with dynamic gates and repair box. In: Proceedings of the Annual Reliablity and Maintainability Symposium, 459-465

[6] Bobbio A, Portinale L, Minichino M, Ciancamerla E (2001) Improving the analysis of dependable systems by mapping fault trees into Bayesian networks. Reliability Engineering and System Safety, 71: 249-260

[7] Bouissou M, Bruyère F, Rauzy A (1997) BDD based fault-tree processing: a comparison of variable ordering heuristics. In: C. Guedes Soares, editors, Proceedings of European Safety and Reliability Association Conference, vol. 3, 2045-2052, Pergamon, ISBN 0-08-042835-5

[8] Bryant R (1987) Graph based algorithms for Boolean function manipulation. IEEE Transactions on Computer, 35(8): 677-691

[9] Buchacker K (1999) Combining fault trees and Petri nets to model safety-critical systems. In: Tentner A., editor, High Performance Computing, The Society for Computer Simulation International

[10] Chiola G, Duthuillet C, Franceschinis G, Haddad S (1991) Stochastic well-formed colored nets and multiprocessor modelling applications. In: Jensen K., Rozenberg G., editors, High-Level Petri Nets. Theory and Application, Springer Verlag

[11] Chiola G, Duthuillet C, Franceschinis G, Haddad S (1993) Stochastic well-formed colored nets and symmetric modeling applications. IEEE Transactions on Computers, 42: 1343-1360

[12] Chiola G, Franceschinis G, Gaeta R, Ribaudo M (1995) GreatSPN 1.7: Graphical editor and analyzer for timed and stochastic Petri nets. Performance Evaluation, (24): 47-68

[13] Codetta Raiteri D, Franceschinis G, Iacono M, Vittorini V (2004) Repairable fault tree for the automatic evaluation of repair policies. In: Conference on Dependable Systems and Networks. Performance and Dependability Symposium

[14] Codetta Raiteri D (2003) Development of a dynamic fault tree solver based on colored Petri nets and graphically interfaced with DrawNET. In: Technical Report TR-INF-2003-10-06-UNIPMN, http://www.di.unipmn.it/Tecnical-R/index.htm

[15] Contini S, Poucet A (1990) Advances on fault tree and event tree techniques. In: A. Colombo G., Saiz de Bustamante A., editors, System Reliability Assessment*,* 77-102, Kluwer Academic Publishers

[16] Contini S (1998) Astra Knowledge Handbook. Logical and probabilistic analysis methods. Special publication of the European Commission Joint Research Centre, 98(138)

[17] Dugan JB, Bavuso SJ, Boyd MA (1992) Dynamic fault-tree models for fault-tolerant computer systems. IEEE Transactions on Reliability, 41: 363-377

[18] Dugan JB, Sullivan KJ, Coppit D (1999) Developing a low-cost, high-quality software tool for dynamic fault tree analysis. Transactions on Reliability, (12): 49-59

[19] Dutuit Y, Rauzy A (1996) A linear-time algorithm to find modules of fault trees. IEEE Transactions on Reliability, 45: 422-425

[20] Dutuit Y, Lemaire O, Rauzy A (2000) New insight on measures of importance of components and systems in fault tree analysis. In: Kondo S., Furuta K., editors, Proceedings of the International Conference on Probabilistic Safety Assessment and Management*,* 729-734, Universal Academy Press, ISBN 4-946443-64-9

[21] Dutuit Y, Rauzy A (1999) New algorithms to compute importance factors CPr, MIF, CIF, DIF, RAW and RRW. In: Proc. of the European Safety and Reliability Association Conference, 1015-1020

[22] Dutuit Y, Rauzy A (2000) Efficient algorithms to assess components and gates importances in fault tree analysis. Reliability Engineering and System Safety, 72: 213-222

[23] Franceschinis G, Gribaudo M, Iacono M, Mazzocca N, Vittorini V (2002) Towards an object based multi-formalism multi-solution modeling approach. In: Proc. of the Second International Workshop on Modelling of Objects, Components, and Agents, 47-66

[24] Hoyland A, Rausand M (1994) System reliability theory, John Wiley & Son

[25] Kovalenko IN, Kuznetsov NY, Pegg PA (1997) Mathematical theory of reliability of time dependent systems with practical applications. Wiley Series in Probability and Statistics, John Wiley & Son

[26] Manian R, Coppit DW, Sullivan KJ, Dugan JB (1999) Bridging the gap between systems and dynamic fault tree models. In: Proceedings Annual Reliability and Maintainability Symposium, 105-111

[27] Manian R, Dugan JB, Coppit D, Sullivan K (1998) Combining various solution techniques for dynamic fault tree analysis of computer systems. In: Proc. Third IEEE International High-Assurance Systems Engineering Symposium, 21-28

[28] Malhotra M, Trivedi K (1994) Power-hierarchy of dependability-model types. IEEE Transactions on Reliability, 43(3): 493-502

[29] Malhorta M, Trivedi K (1995) Dependability modeling using Petri nets. IEEE Transactions on Reliability, 44: 428-440

[30] Natvig B (1985) New light on measures of importance of system components. Scandinavian Journal of Statistics, 12: 43-52

[31] Portinale L, Bobbio A (1999) Bayesian networks for dependability analysis: an application to digital control reliability. In: 15th Conference Uncertainty in Artificial Intelligence, 551-558

[32] Rauzy A (1993) New algorithms for fault trees analysis. Reliability Engineering and System Safety, 40: 203-211

[33] Rauzy A (2001) Mathematical foundation of minimal cutsets. IEEE Transactions on Reliability, 50(4): 389-396

[34] Sahner RA, Trivedi KS, Puliafito A (1996) Performance and reliability analysis of computer systems; an example-based approach using the SHARPE software package, Kluwer Academic Publishers

[35] Sinnamon RM, Andrews JD (1996) Quantitative fault tree analysis using binary decision diagrams. Journal Européen des Systèmes Automatisés, 30(8): 1051-1071

[36] Sinnamon RM, Andrews JD (1997) Improved accuracy in qualitative fault tree analysis. Quality and Reliability Engineering International, 13: 285-292

[37] Schneeweiss WG (1999) The fault tree method, LiLoLe Verlag

[38] Sonza Reorda M, Violante M, Mazzocca N, Venticinque S, Franceschinis G, Bobbio A (2002) A hierarchical approach for designing dependable systems. In: 7th Annual IEEE International Workshop on High Level Design Validation and Test, 63-67

[39] Sullivan KJ, Dugan JB, Coppit D (1999) The Galileo fault tree analysis tool. In: Proc. of the 29th Annual International Symposium on Fault-Tolerant Computing, 232-235

[40] Tang Z, Dugan JB (2004) Minimal cut set/sequence generation for dynamic fault trees. In: Annual Reliability and Maintainability Symposium

[41] Vesley VE (1970) A time dependent methodology for fault tree evaluation. Nuclear Engineering and Design, 13: 337-360

[42] Fussel JB How to hand-calculate system reliability characteristics. IEEE Transactions on Reliability, 24(3)

[43] Vittorini V, Franceschinis G, Gribaudo M, Iacono M, Bertoncello C (2002) DrawNET++: a flexible framework for building dependability models. In: Proc. International Conference on Dependable Systems and Networks

# Index