Linköping Studies in Science and Technology

Thesis No. 1156

High-Level Techniques for Built-In Self-Test Resources Optimization

by

Abdil Rashid Mohamed

INSTITUTE OF TECHNOLOGY
LINKÖPINGS UNIVERSITET

Submitted to the School of Engineering at Linköping University in partial fulfilment of the requirements for the degree of Licentiate of Engineering

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2005

ISBN 91- 85297-90-9

ISSN 0280-7971

Printed by UniTryck, Linköping, Sweden, 2005

High-Level Techniques for Built-In Self-Test Resources Optimization

by

Abdil Rashid Mohamed

April 2005
ISBN 91-85297-90-9
Linköpings Studies in Science and Technology
Thesis No. 1156
ISSN 0280-7971
LiU-Tek-Lic-2005-11

ABSTRACT

Design modifications to improve testability usually introduce large area overhead and performance degradation. One way to reduce the negative impact associated with improved testability is to take testability as one of the constraints during high-level design phases so that systems are not only optimized for area and performance, but also from the testability point of view. This thesis deals with the problem of optimizing testing-hardware resources by taking into account testability constraints at high-levels of abstraction during the design process.

Firstly, we have provided an approach to solve the problem of optimizing built-in self-test (BIST) resources at the behavioral and register-transfer levels under testability and testing time constraints. Testing problem identification and BIST enhancement during the optimization process are assisted by symbolic testability analysis. Further, concurrent test sessions are generated, while signature analysis registers' sharing conflicts as well as controllability and observability constraints are considered.

Secondly, we have introduced the problem of BIST resources insertion and optimization while taking wiring area into account. Testability improvement transformations have been defined and deployed in a hardware overhead minimization technique used during a BIST synthesis process. The technique is guided by the results of symbolic testability analysis and inserts a minimal amount of BIST resources into the design to make it fully testable. It takes into consideration both BIST components cost and wiring overhead. Two design space exploration approaches have been proposed: a simulated annealing based algorithm and a greedy heuristic. Experimental results show that considering wiring area during BIST synthesis results in smaller final designs as compared to the cases when the wiring impact is ignored. The greedy heuristic uses our behavioral and register-transfer levels BIST enhancement metrics to guide BIST synthesis in such a way that the number of testability improvement transformations performed on the design is reduced.

The Swedish Foundation for Strategic Research (SSF) under the INTELECT and STRINGENT programmes at Linköping University supported this work.

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

ISBN 91-85297-90-9 ISSN 0280-7971

Abstract

Design modifications to improve testability usually introduce large area overhead and performance degradation. One way to reduce the negative impact associated with improved testability is to take testability as one of the constraints during high-level design phases so that systems are not only optimized for area and performance, but also from the testability point of view. This thesis deals with the problem of optimizing testing-hardware resources by taking into account testability constraints at high-levels of abstraction during the design process.

Firstly, we have provided an approach to solve the problem of optimizing built-in self-test (BIST) resources at the behavioral and register-transfer levels under testability and testing time constraints. Testing problem identification and BIST enhancement during the optimization process are assisted by symbolic testability analysis. Further, concurrent test sessions are generated, while signature analysis registers' sharing conflicts as well as controllability and observability constraints are considered.

Secondly, we have introduced the problem of BIST resources insertion and optimization while taking wiring area into account. Testability improvement transformations have been defined and deployed in a hardware overhead minimization technique used during a BIST synthesis process. The technique is guided by the results of symbolic testability analysis and inserts a minimal amount of BIST resources into the design to make it fully testable. It takes into consideration both BIST components cost and wiring overhead. Two design space exploration approaches have been proposed: a simulated annealing based algorithm and a greedy heuristic. Experimental results show that considering wiring area

during BIST synthesis results in smaller final designs as compared to the cases when the wiring impact is ignored. The greedy heuristic uses our behavioral and register-transfer levels BIST enhancement metrics to guide BIST synthesis in such a way that the number of testability improvement transformations performed on the design is reduced.

Acknowledgments

First and foremost I would like to thank Almighty God for giving me life, health and ability to study.

I would like to thank my supervisor, Professor Zebo Peng. He has not only given me an opportunity to pursue graduate studies, but also guided me throughout my graduate studies. I am also deeply indebted to Professor Petru Eles for good research advice, fruitful discussions and encouragements.

I would also like to thank all members of the Embedded Systems Laboratory at Linköping University for creating a good working environment, and members of the Swedish Network of Design for Test Research (SNDfT) for fruitful discussions during our regular meetings. Gert Jervan, Alexey Sinelnikov and Lei Zhao have been very helpful. Special thanks should go to Gunilla Mellheden and Lillemor Wallgren for their help in administrative issues. My friends at Ryd you have been there for nice evening discussions after long tiresome days at the University. Peter Hazucha, Odysseas Katapodis, Nasser Ziae, Arnold Rombo, Michael Kwesi Korsah, Sauli Elingarami, Moses Nyangito and Richard Otieno are worth mentioning. Bourhane Kadmiry, Abdelbaki Bougerra and Adam Mantaye thank you very much for sharing your time with me to discuss various academic and general issues.

This work would not have been possible without continuous support from my extended family. I am highly thankful to all my sisters and brothers. Special thanks should go to my mother and my sisters, Rahima and Rauhia, for their continuous advice, encouragement and support. I cannot finish writing this acknowledgement without mentioning my father, late *Rashid*

Mohammed Ali Amour Al-aufy, who played an important role to allow and encourage me to pursue graduate studies.

Finally, I would like to thank my wife, Yusfa Shawwal, for her support and bringing a beautiful daughter, Nassra, into our family.

Abdil Rashid Mohamed

Linköping, March 2005.

The Swedish Foundation for Strategic Research (SSF) under the INTELECT and STRINGENT programmes at Linköping University supported this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

This chapter introduces the thesis topic. It starts by discussing the motivation for the research conducted in this work. After that, a formulation of the research problems is provided. A list of important contributions of the thesis is then enumerated. The chapter concludes by providing an organization of the rest of the thesis.

1.1 Motivation

Testing of electronic chips is an important step of an electronic system's manufacturing process. A Very Large Scale Integration (VLSI) electronic system can only be released into the market if it operates correctly. This is usually ascertained during the testing process.

Modern advances in VLSI technology offer tremendous potential for manufacturing complex electronic systems with multi-million gates. It is now also possible for these complex electronic systems to be implemented on a single chip. On the other hand, testing of such systems is very difficult. Increase in test data volume, reduced access to internal components, and very high operating frequency are some of the challenges that make testing of complex electronic systems a difficult task [68]. To be able to manufacture correctly functioning complex electronic systems, these testing challenges must be well addressed and solved.

Due to the challenges mentioned above, traditional methods are no longer suitable for testing modern, multi-million gate complex electronic systems, such as System on Chip (SoC) or Network on Chip (NoC). Therefore, testing methods must change in order to appropriately utilize the available new capabilities in VLSI technology. Currently, there are several approaches that are proposed to enhance current testing practices. Some of them are briefly discussed in the following paragraphs.

The process of modifying the designs by, for example, adding testing components so that the resulting designs are easy to test, is referred to as Design for test (DfT). Traditionally, design decisions and optimizations regarding silicon area, performance, power consumption, etc. are made first, and then DfT features are added at gate-level. An example of gate level DfT technique is gate-level test point insertion.

Modern DfT techniques are, on the other hand, applied at high levels of abstractions such as register-transfer levels (RTL) or behavioral levels. By raising the level of abstraction at which the DfT techniques are applied, the complexity of designing suitable test infrastructure for the complex systems can be managed. Test synthesis, a technique whereby designing for functionality and designing for testability are integrated together as early as possible in the design process, is one example of high-level DfT techniques. In this way functionality, testability and other issues such as silicon area, performance and power are simultaneously considered and optimized together in the design process. The approach helps to eliminate efforts to re-design for testability later in the design process. In one category of test synthesis, known as high-level test synthesis (HLTS), the tasks of scheduling, allocation and binding are done in such a way that testability is one of the constraints to be satisfied during design optimizations [44], [45], [46].

Built-in Self-Test (BIST) is an approach, which modifies the design in such a way that part of the design is used to test itself [2], [3]. This helps to solve such problems as large test data volume, at speed testing, and test pin limitation. Although BIST is a suitable DfT technique for handling testing problems of the modern complex electronic systems, large hardware and performance

overheads it introduces are the main obstacles to its industrial utilization. Furthermore, inserting BIST components into the designs at gate levels adds to the problem. One way to reduce the negative impact of BIST insertion is to start the insertion of BIST components early during the design process at high levels of abstractions such as register-transfer or behavioral. The technique of considering BIST insertion at these high levels is referred to as BIST synthesis. Therefore, solving the problem of optimizing BIST hardware resources during BIST synthesis process such that constraints like testing time, performance, or power are met is an important step towards improving testability of the complex electronic systems. As VLSI technology advances towards deep submicron implementations, wiring becomes a critical problem. Therefore, BIST synthesis approaches can be even more effective, if they not only consider testability, but also wiring when they modify the designs to improve testability.

The purpose of this work is to address the problems of optimization of hardware overhead during the BIST synthesis process. The hardware overhead includes both BIST resources as well as wiring. The aim is not only to produce designs that are easy to test by using the BIST approach, but also designs that are optimal in terms of total hardware cost including wiring area.

1.2 Problem Formulation

The research done in this thesis concentrates on two main problems:

- BIST synthesis under testing time constraints: The objective is to optimize BIST resources usage under testing time constraints in such a way that testability of all modules in the design is guaranteed. The input to the problem is a design that is represented as a scheduled data flow graph (SDFG) and testing time constraints. The aim is to analyze and enhance the testability of the design in such a way that minimum amount of BIST resources are added so as to simultaneously guarantee the testability and testing time constraints.

- A wiring-aware BIST synthesis: The aim is to find a way to perform BIST synthesis that does not only take into account the cost of BIST resources, but also does quantitative estimation of the wiring cost that is introduced by inserting BIST resources into the design. This approach shall result in more efficient designs in terms of total design area. The design cost considered is the total area of the functional modules, BIST components and wiring area. The problem is formulated as follows: given a design represented as an SDFG along with allocation/binding information, insert BIST modules into the design such that all functional modules are self-testable and the total design area is minimized.

1.3 Contributions

The main contributions of this thesis are the following:

- An approach to optimize BIST resources under testing-time constraints. The approach uses testability analysis results to guide BIST synthesis. Firstly, it explores alternative testability options that exist in the design to help determine which operations can be tested concurrently. Secondly, it proposes design modification heuristics to optimize BIST hardware usage under a given testing time constraint.
- BIST design transformations to guide our wiring-aware BIST synthesis process. We have proposed two approaches to solve this synthesis problem. In the first approach we have formulated the wiring-aware BIST synthesis problem as a simulated annealing optimization problem. In the second approach we have proposed a greedy heuristic for solving the wiring-aware BIST synthesis problem. We have also defined new BIST enhancement metrics that are suited for guiding synthesis towards low hardware BIST overhead.

1.4 Thesis Overview

The rest of the thesis is organized as follows: In Chapter 2 we present some background and related work. In Chapter 3 some of

our specific definitions and concepts, which are used in the discussion of our core concepts, are provided. Chapter 4 concentrates on the approaches and methods to solve the problem of BIST synthesis under testing-time constraints. Chapter 5 addresses techniques to solve the wiring aware BIST synthesis problem. Finally, conclusions and directions for future work are discussed in Chapter 6.

Chapter 2

Background and Related Work

This chapter introduces the background and related works that are necessary for understanding this thesis. It starts by giving an introduction to the design flow of electronic systems. Since the main testing approach targeted by the wok done in this thesis is BIST, background knowledge on BIST and its potential for testing future complex electronic systems are provided. Attempts to integrate testing consideration during high-level synthesis are also discussed. Since testability analysis methods that are able to characterize designs in terms of their testability quality are used in several algorithms and methods presented in this thesis, a discussion of high-level testability analysis is also provided at the end of the chapter.

2.1 Digital Systems Design Flow

Designing and testing of large electronic systems are complex processes. To handle complexity, design activities have been shifted towards higher levels of abstractions.

A synthesis approach for designing digital systems usually starts by specifying a system in terms of its functionality and some implementation constraints [18], [36]. *System-level synthesis* [66] then breaks down a system specification into communicating subsystems (processes). Each subsystem has its own behavioral level specification, which describes its behavior according to the computational problem to be solved.

To transform the behavioral specification to an RTL structural specification, *behavioral level synthesis* is used. In automatic synthesis of VLSI designs, this step is often referred to as *high-level synthesis (HLS)* and it consists of three main tasks [48]. The first task, scheduling, assigns each operation to a time slot. The second task deals with decisions regarding the type and number of functional and memory resources to be used and is known as resource allocation. The third task, known as binding, maps each behavioral operation to a specific functional resource and each variable to a register. The output of the behavioral synthesis is usually given as an RTL data-path and a controller. The data-path consists of interconnected functional modules, such as adders, multipliers, registers, and multiplexers. The controller, at this level, is defined by a state transition table. It controls the data flow in the data-path by providing control signals to the multiplexers, registers and other data-path components.

Some RTL modules are derived from existing libraries and others are designed at logic level. Logic functions are represented by truth tables, algebraic expressions or binary decision diagrams (BDD). During the *logic-level synthesis step*, optimizing and minimizing of logic functions are performed, and a technology independent specification of the system, usually described in terms of large combinational or sequential blocks, is produced. After that, a Boolean expression is transformed by a process known as *technology mapping* into a network of gates from a target library.

According to classical design methods, testing is typically considered after all major high-level design decisions are taken. Consequently, many existing test generation tools work with gate level representations of the designs. At the gate level, designs are quite complex, hence test generation and DfT techniques become a bottleneck of the whole design process. If high fault coverage cannot be achieved, expensive re-design efforts, which can cause delay in product development, are needed to improve testability. Several approaches, which will be discussed in the coming sections, are proposed to address this problem.

In recent years, methods have been elaborated addressing design problems at high levels (register-transfer, behavioral) of

abstractions. At these levels, testability metrics (see Section 2.5) can be used to assess the testability of the designs. DfT techniques assisted by high-level testability metrics can be deployed to embed test components into the designs. It is expected that the embedded test components will make generating efficient test patterns and their application easier.

2.2 Testing of Digital Systems

Manufacturing testing of a VLSI device verifies the correctness of the manufacturing process. Hence, it ensures that the physical device manufactured from the synthesized VLSI design has no manufacturing defects. When testing a VLSI design, it is assumed that the synthesized design correctly implements its original specification. At the end-user or customer environment, testing is used to check for defects that result from wear out, aging or other types of problems. The process of testing digital VLSI systems usually involves two steps.

The first step, known as test pattern generation [12], [24], [61], is mainly a preparatory step. It is executed once during the design process. In this step, test patterns (vectors) together with their expected correct responses are generated. The test patterns and responses will be used to test the VLSI circuit after it is manufactured. The test generation process is usually automated using specialized software tools.

The second step, known as test application, involves the actual testing of the manufactured devices. During the test application process, the test vectors are applied to the hardware device, known as circuit under test (CUT), and the results obtained are compared with the expected correct responses. If the results obtained are different from those expected, then the device is considered faulty.

In a classical testing set up, the test patterns and correct results for testing a given VLSI system are stored in a large testing machine, known as automatic test equipment (ATE). The ATE sends test patterns to the CUT, reads test responses from it and decides whether the CUT is faulty or not.

Testing electronic designs is a costly process. While the cost of manufacturing a transistor has been constantly decreasing, the cost of testing it has remained constant throughout the electronic age [62]. As we go into the deep sub-micron (DSM) era, it is likely that there will come a time when it will be more expensive to test a transistor than to design and manufacture it. The cost of testing a VLSI design can be classified into three main categories:

- In order to simplify testing, it is necessary to modify the design so that the test generation process is able to generate high-quality tests for a given design. These modifications can add area overhead to the chip and can lead to yield reduction. If the modifications for testability are done in the critical path of the design, then a delay overhead is also introduced into the design. This will eventually lead to performance degradation.
- The cost associated with the software processes of test pattern generation. There may also be some costs resulting from the test programming and debugging.
- ATE capital cost and the operational cost of the test centre. This is the cost associated with the manufacturing test.

According to predictions of the testing section of the International Technology Roadmap for Semiconductor (ITRS-Test), the ATE cost dominates product cost [62]. Therefore, use of DfT techniques, and in particular BIST techniques, will continue to grow in order to move test complexity on chip and thus reduce capability requirements and therefore cost of the ATE [62].

Physical defects in VLSI designs manifest themselves at the electrical level as failure modes such as open and short interconnections or parameter degradation [52]. There are many physical defects that can occur on VLSI systems. They are caused by a number of factors such as processing defects (missing contact windows, parasitic transistors and oxide breakdown), material defects (cracks and crystal imperfections), surface impurities (ion migration), time-dependent failures (dielectric breakdown and electro migration) and packaging failures (contact degradation and seal leaks) [1], [34], [52].

To facilitate detection of the physical defects, at the logic and behavioral levels, the failures are modeled as *faults*. Since there are too many real defects and they are hard to classify and analyze, fault models are used to characterize the target faults to be tested. Fault models make it possible to analyze designs with respect to testability in a quantitative manner. There exist many fault models, but here we will mention a few most common ones, which are:

- Single and multiple stuck-at fault models.
- Functional fault model.
- Delay fault model. This is similar to stuck-at fault model, but enhanced with timing information.
- Gate (transition) delay faults.
    - Slow-to-rise or slow-to-fall transitions.
    - Interconnect signal with longer than normal propagation delay.
- Path delay faults – accumulation of gate delay faults over whole paths.

A given fault model, say stuck-at fault, models only a subset of real defects. In this thesis all our discussion will be based on a single stuck-at (SSA) fault model since it is most widely used. Many test generation techniques that target other fault models extend the principles and techniques used in SSA fault model [1]. This fault model assumes that only one line can be faulty at a time. The fault is modeled in such a way that the faulty line is assumed to be permanently stuck-at a logic value 0 or a logic value 1. These faults are assumed to be only present at the inputs or outputs of logic gates such as AND, OR, NAND, NOT and XOR.

Consider an example design depicted in Figure 2.1. According to the SSA fault model this design has five fault sites. Each fault site can be stuck at 0 or stuck at 1. Hence the circuit can have a maximum of ten possible stuck at faults. Generally, a circuit with n nets has $2n$ SSA faults.

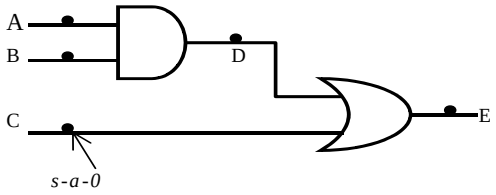


Figure 2.1 Single stuck-at faults

2.3 Automatic Test Pattern Generation

The test pattern generation problem is formulated as follows: given the CUT, usually a gate level net-list, obtain a set of test vectors that will detect a sufficiently large number of the modeled faults in the CUT. As a measure of quality of the test vectors and the test pattern generation algorithms, it is required that the number of test vectors be as small as possible, and high fault coverage, preferably 100%, be achieved. Fault coverage is defined as the ratio of the number of detected faults to the total number of modeled faults.

The cost of test pattern generation depends on the fault model, the complexity of the test generation algorithm and the complexity of the CUT.

Test patterns can be generated at different abstraction levels. A majority of existing tools perform test generation at gate level [24], [61]. With increasing complexity of the designs, test generation at gate level becomes a computationally expensive process. In order to handle complexity issues, hierarchical test generation approaches have been proposed [37]. Such approaches use high-level functional information to speed up the test generation process.

The deterministic test patterns generated by the automatic test pattern generation (ATPG) tools are first stored in an ATE and then used for testing the CUT. In the next section, another approach for generating test patterns and testing designs, known as BIST, will be introduced.

A stuck-at fault at a line l is denoted as l-s-a-v, where v is 0 or 1. A test vector for a given modeled fault l-s-a-v must be able to detect whether the line l is permanently stuck at a value v. To do so the test vector should be able to set an opposite value $\sim v$ at the line l and propagate this value all the way to an observable point. Therefore, the test generation process can be decomposed into two sub-processes: fault activation (excitation) and error propagation. Fault activation is achieved by setting primary input (PI) values that cause a line l to be set to the value $\sim v$. If it is possible to excite a fault then the fault is controllable. Error propagation requires that the error $\sim v$ be transported from the line l to an observable point, usually a primary output (PO). If an error can be observed at a PO then the fault is observable.

Several approaches are used to generate test patterns for a circuit. Test generation methods can be classified as random, pseudo-random, exhaustive or deterministic.

A random test pattern generation algorithm generates test vectors at random without considering the structure of the CUT. For each generated test vector, fault simulation is performed to find all the faults that it can detect. All the faults that can be detected by the generated test vector are then removed from the list of undetected modeled faults. The process of generating test patterns is repeated until all faults are detected or an acceptable level of fault coverage is achieved. Random pattern generation is relatively straightforward, but leads to a test set with many test vectors and usually can have low fault coverage.

More efficient approaches are the deterministic test pattern generation methods, which consider the structure of the CUT when generating test patterns. They can be expensive in terms of computational effort, but they lead to a few, efficient test patterns, which result in a high fault coverage. Generally, a fault-oriented deterministic test pattern generation algorithm works as follows: Pick a fault l-s-a-v whose test is to be generated. Find a way to control line l to an opposite value $\sim v$. Find a way to propagate an error from line l to an observable point, usually a primary output. If it is possible to control the line l to a value $\sim v$ and to propagate the resulting error to the observable point then the test pattern is found. The fault is then removed from the fault list and a test for

another fault is generated in a similar manner. The test generation process terminates when tests for all modeled faults are generated or if the allowed maximum user-defined central processing unit (CPU) time is exhausted. During the course of test generation fault, simulation can also be performed for each test pattern in order to find other faults, which can be detected using the same test pattern.

Generating tests for large circuits is a complex process. The worst-case complexity of the test generation problem is exponential with respect to the number of gates in the circuit [12] and was observed for undetectable faults. Therefore, heuristics are often used to avoid the worst-case complexity of the test generation algorithms. Since heuristics limit the search space in order to run in a reasonable amount of time, they may fail to generate tests for some detectable faults. Examples of deterministic test generation heuristics are PODEM [24], D-algorithm [61] and 9-V algorithm [12] .

A more detailed list of the test pattern generation algorithms is provided in [1] and a performance comparison is available in [4].

2.4 Built-In Self-Test

BIST is an emerging technique for testing complex VLSI systems. To test a design by using a BIST methodology, the design has to be modified (enhanced) in such a way that part of the circuit is used to test the design itself. Therefore, BIST is defined as a DfT technique in which testing is accomplished through built-in hardware components [2], [3].

A general BIST scheme is shown in Figure 2.2. It consists of a test source block, the CUT, a test response analysis block and a test controller block, which manages the application of the tests. In a classical BIST scheme, the test source consists of a special kind of register, test pattern generator (TPG), which generates on-chip test patterns. Recently, a new hybrid BIST approach [38] has been proposed. It enhances the design with a read only memory (ROM) for storing some deterministic test patterns. These stored test

patterns are used to capture faults that cannot be detected by the test patterns generated by the on-chip TPG.

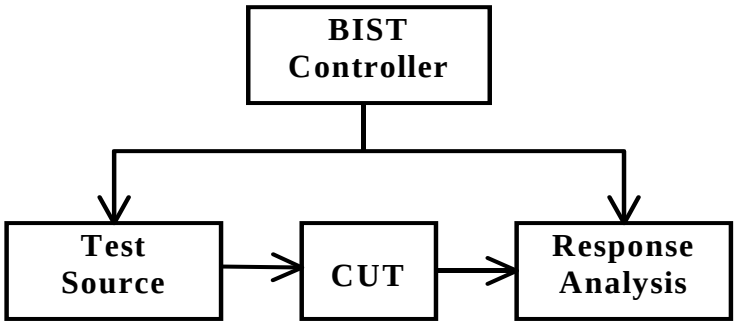


Figure 2.2 BIST Architecture

BIST offers several advantages over classic testing schemes. Since test patterns originate from on-chip sources and test responses are analyzed on-chip, there is no need of huge tester memories to store an enormous amount of test data. Furthermore, the bottleneck problem of sending test data from the ATE to the chip is also addressed. Reducing or eliminating the need for ATE provides field test capability whereby a chip can be tested on field or at customer environment. Another important advantage that BIST offers is the ability to perform at speed testing.

If a testing strategy is well designed, BIST can be used in a hierarchical fashion whereby the same hardware can test chips, boards as well as systems [2]. With BIST the problem of reduced accessibility to internal components is solved since external test data is not needed.

On the other hand, BIST introduces hardware overhead and delay, which can degrade performance.

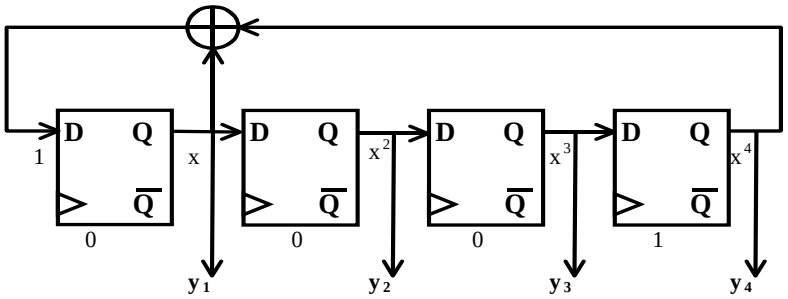
2.4.1 Test Pattern Generation for BIST

On-chip test pattern generators can generate exhaustive or pseudo-random test patterns. Pseudo-random test patterns have many characteristics of random patterns, but are generated

deterministically and hence are repeatable. A cellular automata or hardware based linear feedback shift register (LFSR) [2], [52] can be used to generate pseudo-random test patterns. An LFSR will be generally referred to as a TPG.

An LFSR is a shift register with feedbacks from the last stage and other stages. The outputs of its flip-flops form the test pattern. Each state of the LFSR corresponds to one test pattern. The number of unique test patterns the LFSR can generate depends on the number and location of the feedbacks as well as its initial value, which is known as the seed.

An example of an LFSR is shown in Figure 2.3. It is initialized with the seed *0001*. In the subsequent clock cycles, a series of test



0	0	0	1
1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1
0	1	1	1
1	0	1	1
0	1	0	1
1	0	1	0
1	1	0	1
0	1	1	0
0	0	1	1
1	0	0	1
0	1	0	0
0	0	1	0
0	**0**	**0**	**1**

Figure 2.3 An LFSR example

patterns are produced at the outputs of the flip-flops. This LFSR, which has *n=4* flip-flops, produces a total of 15 (2^n - 1) distinct patterns (except *0000*) as shown in Figure 2.3.

The feedback positions are usually described by a characteristic polynomial [2], [3]. In our example, feedbacks are made from the first (x) and the fourth (x^4) positions, hence the characteristic polynomial of the LFSR is $p(x) = 1 + x + x^4$. The choice of feedback positions (the choice of the polynomial) determines the length of the test sequences generated. Special polynomials known as *primitive polynomials* give maximal length sequences (2^n-1). A polynomial $p(x) = 1 + x + x^4$, which is used in our example, is primitive. It generates a sequence of 15 distinct test patterns before repetition. Therefore, when designing an LFSR a good choice of seed and polynomial is crucial for generating a good sequence of tests.

2.4.2 Test Response Analysis and Compaction

In BIST schemes, all test responses from a CUT are compacted to a single value known as a signature. Thus, one test signature for the whole CUT is obtained at the end of the test application process. Faults are detected by comparing the signature from the CUT and its fault-free signature. If the two signatures are different the circuit is considered as faulty.

Response compaction may result in information loss, which can cause a signature of a faulty circuit to be equal to the fault-free signature. In this way a fault can escape detection. This kind of information loss is known as *aliasing.* Aliasing probability is nevertheless quite small and decreases with increase in the length of the test.

Several compaction schemes exist [1], [52], but the most common one is known as *signature analysis*. A common form of signature analysis methodology uses an LFSR (known here as a signature analysis register) to compact responses from the CUT. At the end of the test application, the content of the LFSR is the signature of the CUT. In its simplest form a signature analysis register is a one input LFSR, which, is suitable for compacting responses from a single output CUT. Mathematical analysis reveals that the LFSR divides the input test response stream, *G(x)*, from a CUT by the

LFSR polynomial *P(x)*, producing an output stream corresponding to the quotient, *Q(x)*, and a remainder *R(x)*. The remainder on the LFSR after completion of the test application corresponds to the final signature of the CUT [1]. Although in this discussion we have presented a single input signature analysis register (SAR), in practice a multiple input signature register (MISR) is usually used to compact responses from the CUT with multiple outputs.

2.4.3 Built-in Logic Block Observer

Built-in Logic Block Observers (BILBOs) and Concurrent Built-in Logic Block Observers (CBILBOs) are other important types of on-chip BIST registers. They are more advanced than regular TPGs or MISRs. While a BILBO can generate test vectors and analyze test responses at different times, a CBILBO can do so at the same time.

BILBOs are needed if it is necessary to improve both controllability and observability of a specific part of the design. On the other hand, if it is necessary to improve controllability, observability and test application time, the CBILBO, which simultaneously performs both test pattern generation and test response compression, can be used. However, BILBOs and CBILBOs usually occupy larger silicon area than TPGs or MISRs of equivalent bit-width.

2.5 High-Level Testability Analysis

To properly synthesize designs for testability, accurate testability metrics are needed. These metrics can be used to rank design alternatives based on their testability during the synthesis process. Testability analysis methods can be classified into three categories: analysis for general testability, analysis for ATPG and analysis for BIST.

There has been work done on testability analysis at gate level, register-transfer (RT) level and behavioral level. In this thesis we are mainly interested in RT and behavioral levels testability analysis approaches.

2.5.1 Register Transfer Level Testability Analysis

At RTL there is a number of testability analysis works done.

Chen and Menon [13] proposed a probability based testability analysis approach. They defined *0-combinational controllability (CC_0)* as the probability that a signal has a value 0. Similarly, *1-combinational controllability (CC_1)* is defined as the probability that a signal has a value 1. *Sequential controllability* is an estimate of the length (time steps) of a sequence for setting a signal in a circuit to a specific value. 0-sequential controllability and 1-sequential controllability are defined. *Combinational observability* is the probability that a change in the input will result in a change in the output. *Sequential observability* is an estimate of the number of time frames required to propagate the effects of a signal change on a line to the primary output.

Gu [28] has also proposed some probability based testability metrics. For each line in the design he associated four testability values, namely combinational controllability, combinational observability, sequential controllability, and sequential observability. He also defined a notion of controllability transfer factor to describe how the controllability of the input and output of functional modules are related. The controllability transfer factor is the probability of setting a value at a module's output by randomly exercising its inputs. Similarly, the probability of observing a module's input by randomly exciting it's other inputs and observing its output is known as observability transfer factor. These metrics target improvement of ATPG and are used in a testability analysis and enhancement algorithm. Designs are described using an extended timed Petri net (ETPN) [56].

Papachristou and Lai [55] proposed testability metrics based on the notion of entropy. The metrics are computed for registers and are suitable for pseudo-random BIST testing. Controllability of a register is measured by the metric known as *randomness,* which is defined as the ratio of its output entropy to the maximum output entropy. Observability is measured by the metric called *transparency*, which is defined as the probability that an arbitrary change in the signal value can be observed at the primary output. These metrics are computed for the variables of a scheduled data flow graph (SDFG). For a register w, which implements two SDFG variables a and x, the testability metrics are computed as follows:

$$R_w \approx C_1 \frac{R_a + R_x}{2} + C_2 \frac{M_a + M_x}{2} + C_3 \text{ and } T_w = \frac{T_a + T_x}{2}$$

where R_a, R_x and R_w are the respective randomness of the variables a, x and w. Coefficients C_1, C_2, C_3 depend on bit length. M_a and M_x are the probability distributions of a and x. T_a, T_x and T_w are the respective transparencies of a, x and w.

Flottes et al. in [19] proposed probability based testability metrics and a testability analysis method and used them during a HLS process. They concentrated on testability bottlenecks induced by reconvergence or module transparency properties. The *Controllability measure* of a module N is defined as $C(N)=y/2^n$, where y is the number of any patterns (test or not) that can be propagated to the node N from the primary inputs. This is likened to the probability of propagating test patterns to the node N. The controllability transparency coefficient T_c is the ratio of different values that can be set on the functional unit output over 2^m, where m is the bit-width. To take into account the correlation between input ports, the controllability transparency coefficient is defined as

$$T_c = \frac{(C_2 - C_1) \times p}{n} + C_1$$

where n is the bit-width of the input ports, p is the number of common bits between the input ports, C_1 is the proportion of values that can be obtained on the functional unit output given that its input ports are not connected to each other ($p=0$) and C_2 is the proportion of values that can be obtained on the functional unit output given that its input ports are connected to each other ($p=n$).

The *observability measure* is defined as $O(N)=y/(2^{n-1}\times(2^n-1))$, where y is the number of pairs of different responses that can be propagated and differentiated at the primary outputs regardless of their values, and $(2^{n-1}\times(2^n-1))$ is the total number of possible pairs. $O(N)$ is the probability to differentiate fault-free and faulty responses of the node N. the observability coefficient T_o is the proportion of pairs of input values that can be distinguished on the isolated functional unit output. For a left side shifter with n

input bits and n output bits, and least significant bit set to 0, the obsevability transparent coefficient T_o is given by

$$T_o = \frac{2^n - 2}{2^n - 1}.$$

2.5.2 Behavioral level Testability Analysis

Boubezari et al [11] proposed a high-level testability analysis and test point insertion based on analysis and modification of very high-speed integrated circuit hardware description language (VHDL) specifications. They target scan based BIST. Their testability analysis method searches for hard to detect bits of signals and variables that are explicitly declared or implied in the VHDL specification. Test point insertion is carried out using overloaded VHDL functions and procedures. Since the original VHDL specification and the added test point VHDL code are simultaneously synthesized, the approach can lead to very efficient scan based self-testable designs.

Larsson [41] proposed a technique that analyzes the testability of the behavioral VHDL specifications. Testability properties are extracted by analyzing *variable range, operation testability and statement reachability*. Value range of a variable v, *VR(l,v)*, at line l is the range of valid values the variable can have as given in the specification. *defVR(v)* is the full range of values defined for the variable v. A relative value range, *RVR*, for a variable v at line l is defined as the ratio

$$RVR(l, v) = \frac{|VR(l, v)|}{|defVR(v)|}.$$

Operation testability, *opT*, captures the change in distribution of test vectors in the output of an operation assuming all possible test vectors on its input.

$$opT = \frac{1}{Q(op)^{(1/b)}}$$

In the equation above, *b* is the number of bits in the inputs and *Q(op)* is the difference between the distribution on the output of an operation and the uniform distribution.

These testability metrics do not target any particular DfT technique, but have been used for guiding the selection of partial scan registers. They were also used in guiding testability improvement transformations applicable directly in the VHDL specification [42].

2.5.3 Symbolic Testability Analysis

Bhatia and Jha [9], [10] proposed a hierarchical testability analysis approach. Ghosh et al [22], [23] further extended the work and introduced the so-called symbolic testability analysis (STA). STA derives control (justification) and observation paths for all operations in the design, which is represented as an SDFG. The designs analyzed by STA can be tested either using an ATPG based approach or BIST. If ATPG based testing is targeted, all justification and propagation paths are computed with respect to the primary inputs and primary outputs. If BIST testing is targeted, all justification and propagation paths are computed with respect to built-in TPGs and MISRs.

STA approach can be applied to designs represented at the behavioral or RT levels. If a design is specified at the behavioral level, an SDFG is used for testability analysis. On the other hand, if a design is specified at the RTL, then controller and data path are used to extract an intermediate test control data flow (TCDF) [22] graph representation. In such cases STA uses the TCDF to perform testability analysis of the design. Therefore, depending on the available design representation, STA can use an SDFG or a TCDF graph for performing testability analysis.

To analyze the testability of the design, for each operation, STA searches the SDFG to find a set of justification paths from PIs (or TPGs) to the operation in order to get direct accessibility for bringing test patterns to the operation. Similarly, STA searches the SDFG to find propagation paths from the operation to POs (or MISRs) for observing test responses.

The justification paths for the operation under test are obtained by looking at its inputs and tracing back the paths that can be used

to set its values from the PIs (or TPGs). Similarly, propagation paths (paths needed to propagate test responses to POs or MISRs) are obtained by looking at the operation's output and trace forward the paths to the POs or MISRs. To derive these paths, it is necessary to force intermediate active variables (registers) to take particular values to assist in transporting test data through the circuit. Suppose that test data is to pass through an intermediate operation with inputs *x*, *y* and output *z*. If test data shows up at the input *x* of the intermediate operation, then the other input, *y*, is assigned an appropriate *neutral or identity* value in such a way that test data will be transmitted unchanged to the operation's output, *z*.

To derive justification and propagation paths for all operations, STA defines a number of Boolean values for controllability and observability of each SDFG variable. General controllability, $C_g(v,n)$, of an SDFG variable v on the n^{th} control cycle is the ability to control it to any arbitrary value from the corresponding primary inputs or TPGs. Similarly, controllability of a variable to the constant value *1*, $C_1(v,n)$, controllability to the constant value *0*, $C_0(v,n)$, and controllability to a vector of all *1*'s, $C_{a1}(v,n)$, are defined. $C_1(v,n)$ represents an identity value to be set on one input of the multiplier operation so that the test data on its other input can reach its output unchanged. $C_0(v,n)$ is a neutral value, which if set on one input of the addition operation, test data on the other input can reach the output unchanged. Similarly, $C_{a1}(v,n)$, helps to pass test data through the logic AND operation. Observability, $O_v(v,n)$, of a variable v on the n^{th} control step is the ability to observe any value of the variable at a primary output or on-chip MISR.

The derivation of justification and propagation paths for a given design will be discussed in more details in Section 3.3.

2.6 High-Level Synthesis for Testability

Very good designs that have been optimized at high-level for area and delay only, can have bad testability. Improving testability by gate level techniques can introduce large area and performance overhead, which can impair the designs that are already optimized

for area and delay. This is because the DfT technique's searching for testable designs is restricted by the already chosen RTL architectural alternative.

HLS optimizes a circuit by exploration of alternative solutions for scheduling, allocation and binding. Thus, a HLS system explores a large design space. Some of the high-level architectural alternatives are more testable than others. Therefore, if the HLS system takes testability as one of the constraints to be satisfied, then a testable architecture may be found very early during the design process. Consequently, the need to improve testability at a later stage will be reduced and the resulting RTL design may need very small gate-level DfT modifications.

High-level test synthesis (HLTS), also known as high-level synthesis for testability, refers to the inclusion of DfT techniques in a high (behavioral, RT) level synthesis process. This means that during scheduling, allocation and binding, testability is also taken into account as one of the design attributes along with area and performance. In short, test synthesis focuses on how and when in the synthesis process

I. Constraints are placed to obtain easily testable circuits.

II. DfT structures are incorporated into the design [52].

An important task that a HLTS system needs to do is to predict which high-level structures can make testing hard. Examples of such structures are self-adjacent registers and loops, which are considered undesirable from the testing point of view. Creation of these structures should, therefore, be avoided during the HLTS process.

Testability can be considered at all abstraction levels, although the effectiveness can be different. In literature, there are HLTS approaches that enhance testability at both, behavioral and register-transfer levels. HLS tasks that can be explored to enhance testability are scheduling, allocation, binding and partitioning.

In order for the HLTS approach to be successful, there must be a way to measure or predict testability while still at a high-level of abstraction. As a result, a more testable design can be selected during the synthesis process. There are many approaches, which

can be used to characterize testability of the designs at high-levels of abstraction. These approaches have developed *high-level testability metrics* (see Section 2.5) that can be used to accurately measure design testability during the synthesis process. Consequently, the metrics can be used to guide choices of suitable test insertion points to improve controllability and observability. If BIST is used, control test points are then connected to TPGs and observability test points are connected to MISRs.

During HLTS another question is what to optimize and what are the constraints to be satisfied. Usually, many test synthesis approaches try to optimize area, delay, or testability.

In this section, we will discuss several HLTS techniques, which are proposed in the literature: HLTS for general testability, HLTS for BIST, HLTS for scan and test point insertion, and other approaches.

2.6.1 Synthesis for General Testability

These approaches do not target any specific testing strategy. They simply eliminate undesirable structures or introduce generic features that are friendly to any testing mechanism. A few examples of the existing approaches will be discussed below.

An allocation approach to improve testability without assuming any specific testing strategy was proposed in [46]. The approach has developed allocation rules for test synthesis. The first rule is used to enhance controllability and observability of registers during the register allocation process. In that work it is assumed that a register is directly controllable if at least one PI variable is assigned to it. Similarly, a PO variable assigned to a register makes it directly observable. Thus, controllability and observability can be improved by maximizing the number of registers which are assigned at least one PI/PO variable. This testability aware register allocation is summarized as a synthesis rule number one (SR1), which states that “whenever possible, allocate a register to at least one PI or PO variable”.

To further improve testability, the approach in [46] also minimizes sequential depth during register and module allocation. This is summarized in a synthesis rule number two (SR2), which states that “reduce sequential depth from an input register to an output

register". The two rules were implemented in a register allocation algorithm.

Since presence of loops in the circuit makes it hard to test, one possible HLTS task is to eliminate or reduce sequential loops during the allocation process. Loops in the RTL designs are created as a result of loop constructs in the behavior or as a result of hardware sharing. For acyclic data flow graph (DFG), register and module sharing performed during allocation can create sequential loops. Thus, an allocation scheme that avoids creating sequential loops by introducing sequential paths to an input output (IO) register can improve testability. Cyclic DFG specifications with loops (while, for, etc.) can also create sequential loops in the circuit. These loops can be avoided by breaking the loops in the behavior. This can be achieved by assigning variables on the cyclic data flow to an IO register or a known scan register so as to ease controllability and observability of the sequential loop. This synthesis heuristic is summarized in synthesis rule number three (SR3) which aims at reducing sequential loops. It states that "do proper resource sharing to avoid creating sequential loops for acyclic DFG, and assign IO registers to break the sequential loops in a cyclic DFG" [44], [46].

If allocation for testability is performed after scheduling has been decided, its effectiveness depends on whether scheduling produces a good schedule, which is able to produce register/module allocations that can enhance controllability/observability and reduce sequential depths and loops. Therefore, there is a need for a testability aware scheduling, which performs scheduling for controllability/observability enhancement and for sequential depth/loop reduction. Schedules obtained from such an approach can make the successive synthesis tasks (allocation, binding) more effective in finding efficient testable designs.

A testability aware scheduling is proposed in [45]. This approach assumes that it knows in advance the allocation algorithm that will be used. Hence, it performs scheduling in such a way that the resulting schedules will favor the targeted allocation algorithm to find testable designs. It is summarized in synthesis rule number four (SR4) which states that "schedule operations to support application of SR1, SR2, and SR3". This scheduling rule is

embedded in a testability aware mobility path-scheduling (MPS) algorithm [45]. It is an iterative/constructive scheduling algorithm that iteratively performs partial scheduling followed by a testability analysis procedure, which applies the synthesis rule SR4.

2.6.2 Synthesis for BIST

The importance of using BIST has already been emphasized in Section 2.4.

A *high-level BIST synthesis* approach inserts BIST structures into the design during the synthesis process. Considering BIST insertion, at high-levels of abstraction, does not only help to tap advantages of BIST, but also helps to reduce hardware overhead and performance degradation that BIST introduces. Since high-level synthesis for BIST optimizes area, delay and BIST at the same time, the resulting designs are likely to be more efficient than those that are first optimized for area and delay, and later, at lower levels, for BIST. Several high-level BIST synthesis approaches have been proposed in the literature. The basic idea implemented by these BIST synthesis works is to make a selection of functional registers that will be converted to BIST registers such as LFSR, MISR, BILBO or CBILBO. In order to synthesize efficient designs, these approaches also reduce structures that are bad for BIST. An example of such a structure is a self-adjacent register. A self-adjacent register demands to be converted to a TPG and a MISR at the same time. Since this can only be achieved by converting it to a CBILBO, which is very expensive in terms of area overhead, self-adjacent registers should be avoided.

The Syntest HLTS system [30] utilizes the freedom in allocation to synthesize self-testing designs. The system completely avoids creation of self-adjacent registers and guarantees testability by using BILBOs. To reduce BILBO overhead it also exploits circuit functionality. Testability of each operation is guaranteed by mapping it to a testable functional block (TFB) such that the output of a given TFB cannot drive any of the inputs of the same TFB. Thereafter, compatible resources, which cause no conflict and do not introduce self-loops, are mapped to TFBs to minimize area and delay. During mergers, functional testability metrics (randomness, transparency) are used to help remove unnecessary

TPGs, MISRs and BILBOs without impairing fault coverage too much.

An approach that minimizes the number of self-adjacent registers, after performing scheduling and module allocation, is proposed in [6]. This approach, known as RALLOC, imposes testability constraints during register allocation. In doing so, BIST area overhead is reduced. The synthesis problem is modeled as register conflict graph, which is solved by a graph coloring method to find a register allocation with minimal number of self-adjacent registers. The remaining self-adjacent registers are converted to CBILBOs and non-self adjacent register to BILBOs. The resulting RTL design is evaluated by a cost function,

$$cost_{BIST} = 20nsa\text{-}reg + 35sa\text{-}reg + \#mux\text{-}in + \#int + \#ctl$$

where *nsa-reg* is the number of non-self adjacent registers, *sa-reg* is the number of self-adjacent registers, #*mux-in* is the number of multiplexer inputs, #*int* is the number of interconnects and #*ctl* is the number of control signals.

Syncbist, an approach to synthesize self-testable RTL data paths with high degree of testing concurrency is proposed in [31], [32] and [33]. The approach uses partial intrusion BIST (some functional registers are used as TPGs, MISRs or both). Test paths (paths through which test data propagate) are identified for testing each module. The paths are scheduled to minimize testing time. If two or more test paths share hardware, then they are said to be in conflict and must be scheduled in different test sessions. *Hard conflict* occurs if the same register is used as a TPG in one test path and a MISR in another. Though, hard conflicts can be resolved using expensive CBILBOs, the Syncbist approach decides to schedule the conflicting paths in different test sessions. The maximum number of hard conflicting paths is the lower bound on the number of test sessions. Two test paths that share the same intermediate functional registers, modules, multiplexers or interconnections at the same cycle are said to be in *soft conflict*. Soft conflicts are resolved by scheduling a shared resource in different cycles. In order to maximize test concurrency, test conflicts are avoided during the HLS process. The synthesis process is assisted by testability metrics (conflict probability and coverage probability).

An approach to minimize testing time in a combined BIST and ATE environment was presented in [63]. However, the issue of sharing BIST circuitry among cores or functional modules was not studied. The work did not explore parallelism inside the cores to reduce test time during high-level synthesis either. An efficient approach for BIST hardware insertion with short test application time is proposed in [53], [54]. It achieves concurrent testing of modules by sharing test pattern generators. Both short test application time and low BIST overhead are achieved, but BIST insertion is performed without testability analysis and loss of randomness of test data may happen when some modules are deeply embedded in the design.

An integer linear programming (ILP) formulation for making simultaneous trade-off between test time and BIST resource optimization is proposed in [47]. The approach results in very high BIST hardware overhead and test time minimization is neither sufficiently discussed nor supported by experimental results.

Chen [13] proposed an approach for concurrent test scheduling in a BIST environment. First, he assigned BIST registers to each circuit under test (CUT) and then efficiently solved the test-scheduling problem to minimize test time and improve BIST register utilization. BIST register selection is performed without testability analysis; hence no optimal procedure for selecting BIST registers is given. Furthermore, selection of BIST registers and test scheduling are independently performed.

Kim [39] introduced an approach to find an optimal register assignment for testing a design in a given number of test sessions.

2.6.3 Synthesis for Scan and Test Point Insertion

A scan register is a modified shift register that can scan-in test vectors and scan-out test responses. By scanning test data in and out they can provide direct controllability and observability of registers. Therefore, one way to improve testability is to convert all registers into scan registers and arrange them in a large chain known as scan chain. This is a full scan approach. An alternative is to make an intelligent selection of a few registers to be converted to scan registers such that full testability is achieved. Converting

only a selected set of registers to scan registers introduces low overhead and is known as partial scan approach.

Another approach to improve controllability and observability is through test point insertion, whereby hard to control parts are identified and then connected directly to controllable points (primary inputs) and hard to observe parts are connected to observable points (primary outputs).

Genesis [9], [10] is a synthesis for hierarchical testability approach, which uses a hierarchical representation of a design to speed up justification and propagation of test data. This means that the test set for each module under test (MUT) is computed at gate level assuming that the module is connected to PIs and POs. During HLS behavioral/RTL information is used to search for test transfer paths for transporting test data from PIs to the MUT and from the MUT to POs. In Genesis, a hierarchical testability analysis identifies propagation and justification paths for each module in the SDFG. An allocation algorithm, which is integrated together with the hierarchical testability analysis, synthesizes the data path and controller. The synthesized design has a complete system level test set. Modules without test paths are enhanced using test multiplexers, which are used to enhance controllability or observability. The control flow in the SDFG is used for value justification and propagation; hence scan technique is not needed. The allocation algorithm used in Genesis works on a compatibility graph constructed from the SDFG. The nodes are variables and operations. Weights on edges represent preference of assigning the two corresponding variables/operations to the same registers/module in such a way that area will be reduced. Connected nodes with highest weight are merged during allocation. If the partially allocated design is hierarchically untestable then the last allocation is undone, and another allocation is tried. If all possible allocations yield an untestable design, a multiplexer is added to enhance testability of a given module.

A test point insertion methodology at RTL is proposed in [21], [22], [23] by Ghosh et al. Hierarchical testability analysis is used to identify untestable modules and a small number of test multiplexers is inserted into the RTL design to make it

hierarchically testable. To minimize delay overhead, whenever possible, insertion of test multiplexers on critical paths is avoided.

The test synthesis system proposed in [17] takes a control dataflow graph (CDFG) as an input and breaks the loops with a minimal number of scan registers. During the HLS process sequential loop creation is avoided by sharing scan registers.

Gu *et al.* [28] proposed a test synthesis approach based on testability analysis and improvement of VHDL specifications. Testability improvement is achieved through partial scan and test point insertion.

2.6.4 Other Approaches

Several other high-level test synthesis approaches are presented in the literature.

Bhattacharya *et al.* [8] proposed a transformation and re-synthesis for testability approach.

In [40] Kim *et al.* used testability measures based on controllability/observability of registers, the length of sequential depth, and the number of sequential loops to do simultaneous scheduling and allocation for testability. They refer to their approach as a stepwise refinement synthesis for easy testability.

Partitioning for testability is a divide-and-conquer approach that simplifies testing. Partitioning can be useful to remove redundancy, make partitions fan-out free, guide breaking of feedback loops, insert test points at hard to test places and help to remove random pattern resistance [26], [27]. After partitioning is done at high levels, the resulting partitions can then be synthesized together. The partitions are able to share testing components whereas tests for each partition are separately generated.

Since generating test patterns for circuits with long feedback loops and long sequential depth is very hard, a re-synthesis (re-scheduling, re-binding) approach to avoid loop creation caused by hardware sharing is proposed in [57].

During register allocation and binding, not only the number of registers is minimized but also some constraints are imposed to

increase controllability and observability of registers. In [67] Yang *et al.* proposed merger transformations to improve testability. Their idea is to merge nodes with good controllability with nodes with bad observability and nodes with bad controllability with nodes with good observability. The overall impact is to have nodes with both good controllability and good observability.

2.7 Wiring and Interconnect Issues

In deep sub-micron implementation, wiring can take substantial amount of the total chip area. With the development of the microelectronics technology, there is a clear trend towards deep sub-micron implementation, where the interconnecting wires dominate the silicon area cost, and it is even more important to consider the wiring effect in the future deep sub-micron VLSI implementations.

Since exact wiring information is only available after physical design steps, such as floor planning and placement, are performed, most of the existing high-level BIST synthesis and other test synthesis approaches usually do not consider wiring effect. The approaches presented in [6], [11], [16], [21] - [23], [31] - [33], [53], [54], [58], [59] and [63] - [65] are examples of BIST works that do not consider the impact of placement of the functional and BIST modules in the final design. Consequently, they lead to designs, which are optimal in terms of the numbers of functional modules and BIST resources, but take more silicon area to implement since the interconnections take a lot of silicon space. Therefore, it is important to take floor planning and wiring cost into account during the BIST synthesis process.

To get area efficient designs, the impact of wiring area contribution should be addressed as early as possible so that functional, BIST and wiring area can be simultaneously optimized. In this way, the resulting designs are likely to be better in terms of total area as compared to the case when wiring is ignored during the synthesis.

In order to take wiring information of the final designs into account during the BIST insertion process, it is needed to predict the wiring area and lengths at higher levels of abstractions. There

has been some work done in this direction. Alvandpour et al. [5] have developed a heuristic to estimate wiring lengths at RT or higher levels of abstraction. The approach was later deployed by Hallberg et al. [29] to predict area increase due to wiring in a high-level synthesis system under local timing constraints. Their approach makes use of a few technology dependent parameters, which can be extracted from technology libraries. Recently, Goel and Marinissen [25] have proposed a model of wiring-length computation for core based system-on-chip testing, where they have assumed the layout of the modules to be known beforehand.

2.8 Summary

A general overview on how digital systems are tested has been provided. We have also presented a number of approaches that are used to synthesize easy to test systems. Many of these methods make use of testability analysis and testability metrics. We have presented a number of testability metrics. Some of them are based on probabilities/entropies and others are based on analysis of VHDL specifications. There are also methods, which search the justification and propagation paths based on the structures of the SDFGs. High-level testability metrics and testability analysis are helpful in detecting hard to test parts of the designs early during the design process. This can help to accelerate the synthesis of easy to test designs.

Despite the fact that the presented testability analysis approaches can be used to assist synthesis of designs that have very high fault coverage and low hardware overhead, the trade-off between hardware overhead and test time at high-level is insufficiently explored. Most of the current approaches to BIST test time minimization are based on test scheduling optimizations, but efficient test time minimization by sharing BIST components is not well addressed. Similarly, minimizing BIST hardware overhead under testing-time constraints needs further investigation.

We have motivated the need for considering wiring cost during the BIST synthesis process as VLSI technology advances towards DSM era. In our literature survey, we have found out that many of the existing test synthesis approaches have omitted wiring issues.

This thesis addresses the weaknesses and limitations of the current approaches to make them more suitable for testing DSM VLSI circuits. In Chapter 4 we address the trade-off between BIST overhead and testing time. In that chapter we provide an approach, which uses testing time as a constraint in a BIST hardware minimization problem. In Chapter 5 we propose a solution to the problem of a wiring-aware BIST synthesis.

Chapter 3
Preliminaries

This chapter focuses on concepts and definitions that are used in the rest of this thesis. The chapter starts by describing the design representation for our algorithms. Then our definitions of testability of SDFG nodes and register-transfer modules are provided. The concept of alternative test environment options and how they can be shared for testing the design is also discussed.

3.1 Design Representation

3.1.1 Behavioral representation

The input to our BIST optimization technique is an internal representation of the behaviors based on a scheduled dataflow graph.

Definition 3.1: A data flow graph (DFG) is a directed graph $G=(V,E)$ which consists of a finite set of nodes $V=\{v_1, v_2 \ldots v_n\}$ and an asymmetric data flow relation whose elements are directed edges $E \subseteq V \times V$.

Nodes in the DFG represent operations and edges represent data flow relation. A directed edge $v_i \rightarrow v_j$ from $v_i \in V$ *to* $v_j \in V$ exists if the data produced by the operation o_i (represented by node v_i) is consumed by the operation o_j (represented by node v_j).

Definition 3.2: A schedule of a DFG $G=(V,E)$ is a mapping $S:V \rightarrow \{1,2,\ldots,L\}$ where for any pair of operations $o_i, o_j \in V$, $S(o_i)<S(o_j)$,

if $(o_i,o_j) \in E$. $\{1,2,\ldots,L\}$ corresponds to control steps. L is the latency of the schedule.

Definition 3.3: A scheduled data flow graph (SDFG) is a DFG with scheduled operations that obey data dependencies and have a notion of control steps.

An example of an SDFG is shown in Figure 3.1a, where horizontal dashed lines are used to delimit control steps.

Throughout this thesis, unless otherwise stated, our testability analysis is always performed on the SDFG representation of the designs.

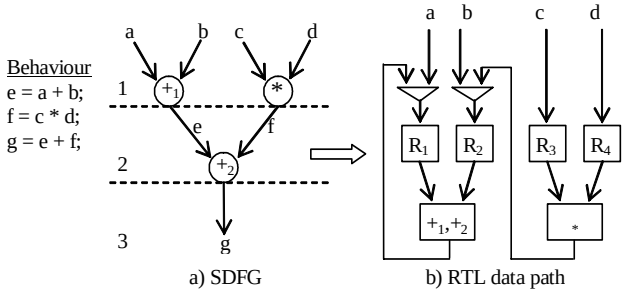


Figure 3.1 An SDFG and its corresponding RTL data path

3.1.2 Register-Transfer Level Structural Representation

The structure of a design at the RTL consists of two parts: a data path and a controller. The data path deals with data manipulation and the controller controls flow of data in the data path units in such a way that the implemented behavior is executed.

The data path consists of three main types of components.

- Functional modules such as adders, multipliers, and ALUs, which perform arithmetic and logic operations specified in the behavior.
- Memory units such as registers, RAMs, and ROMs, which store values of variables and constants generated and consumed during execution of the operations.

- Communication units such as multiplexers and buses, which enable transfer of data between the functional modules and memory units.

An example of the data path, which is also represented as a graph, is shown in Figure 3.1b. It is an implementation of the SDFG in Figure 3.1a.

This thesis concentrates on improving testability of the data path by using the BIST strategy.

3.2 Testability of SDFG Operations and Variables

In this thesis we assume that BIST is used as the basic testing strategy. Definition of testability of SDFG variables and operations is based on the use of symbolic testability analysis (STA) [21], [22], [23]. STA asserts an operation to be testable if there is a guaranteed transparent path from on-chip TPGs to the inputs of the operation for supplying test patterns, and a transparent path from the output of the operation to an on-chip MISR or BILBO for observing test results. In other words, an operation is testable if its input operands are controllable and its output observable at the same time.

To model testability and do testability analysis of the design, we have defined a number of concepts.

Definition 3.4: A primitive STA value (PSTAV) of a given SDFG variable v is its value in a given control step n at which it exists.

Since STA analyzes testability of the SDFG by using its functional behavior, only a few PSTAVs have testability importance. We denote these values as *gPSTAV, 0PSTAV, 1PSTAV, a1PSTAV,* and *rPSTAV. 0PSTAV* is the value *0, 1PSTAV* is the value *1, a1PSTAV* is an all *1* vector and *gPSTAV* is any arbitrary value which can be requested. *gPSTAV* is used to model a test pattern and *rPSTAV* is used to model a test response. When analyzing the testability of a design it will be needed that a test pattern is set at a given variable. Therefore, to set the value of a variable to a *gPSTAV* is to

set it to a test pattern. The value of a variable is set to *rPSTAV* if the variable stores a test response. A set consisting of those STA values which are needed for testability analysis is called the *set of testability important PSTAVs* and is denoted as *STIPSTAV*. Thus,

$$STIPSTAV = \{ 0PSTAV, 1PSTAV, a1PSTAV, gPSTAV, rPSTAV \}$$

To model testability, we have defined a number of basic Boolean STA constraints (BSTAC). We use them to formally define the way we model concepts we have used for doing testability analysis.

Definition 3.5: *g-Controllability constraint*, read general controllability and symbolized $C_g(v,n)$, of an SDFG variable v on the control step n is the ability to set (control) the value of v to *gPSTAV* in control step n from the PIs or TPGs. If this ability can be achieved then *g-Controllability* is true and the variable v is controllable to a value *gPSTAV*, otherwise it is false and v is not controllable to *gPSTAV*.

Similarly, controllability to the constant value *1 (1-Controllability),* controllability to the constant value *0 (0-Controllability)* and controllability to a vector of all *1*'s (*a1-Controllability)* have been defined.

Let us assume that a notation $C_\alpha(v,\beta)$ denotes an STA constraint and means that a variable v is controlled to a value α in a control step β. With this notation, for a given variable v in control step n, its *0-Controllability* will be denoted as $C_0(v,n)$, *1-Controllability* as $C_1(v,n)$, *g-Controllability* as $C_g(v,n)$, *r-Controllability* as $C_r(v,n)$ and *a1-Controllability* as $C_{a1}(v,n)$.

Definition 3.6: BSTACS of a variable v in control step n is defined as a set consisting of all basic STA constraints, that is

$$BSTACS = \{C_0(v,n), C_1(v,n), C_{a1}(v,n), C_g(v,n), C_r(v,n) \}.$$

Definition 3.7: A test environment (TE) is defined as a set consisting of any combination of basic STA constraints in the design.

Since a TE consists of constraints which can be satisfied or not, we define an evaluation of a TE as a Boolean product of all the elements (STA constraints) in the TE, i.e.

$$eval(TE) = \bigwedge_{i=1}^{|TE|} x_i, x_i \in TE \qquad (3.1)$$

After the TE and its evaluation are clearly defined, we will now define controllability TE of a variable (vCTE), controllability TE of an operation (pCTE), observability TE of a variable (OTE), testability TE of a variable (vTTE) and testability TE of an operation (pTTE).

Definition 3.8: vCTE of a variable v in control step n (vCTE(v, n)) is a set of basic STA constraints which together can set the variable v to a primitive STA value *gPSTAV* from PIs or TPGs. In other words, vCTE(v,n) is a TE whose constraints together can set the variable v to a value *gPSTAV*.

An SDFG variable v is controllable if *eval(vCTE(v,n))* is true.

Definition 3.9: pCTE of a 2-input operation, *p,* in control step n (pCTE(p,n)) is the union of the variable controllabilities of its inputs x and y, i.e.

$$pCTE(p,n) = vCTE(x,n) \bigcup vCTE(y,n) \qquad (3.2)$$

An operation, p, is controllable if *eval(pCTE(p,n))* is true.

To read test results from an operation, which is being tested, we need to be able to transport the test response (rPSTAV) from the tested operation's output through intermediate operations and variables to a variable that can be read directly. A variable, which can be read directly because it is mapped to a PO, MISR or BILBO is an observable variable. Suppose we are given a variable v_i in control step n_i, which needs to be observed and a variable v_j which is observable in control step n_j. Analysis of the observability of v_i at v_j can be converted to a problem of setting a primitive STA constraint ($C_r(v_i,n_i)$ from v_i to v_j. This is similar to the controllability analysis problem of finding a set of basic STA constraints which together can set the variable v_j in control step n_j to a primitive STA value *rPSTAV* from the variable v_i.

Definition 3.10: OTE of a variable v_i set at a value *rPSTAV* in control step n_i at another variable v_j in control step n_j is a set of STA constraints which together set the value of v_j in control step n_j to *rPSTAV* from v_i. It is denoted by $Obv(v_i, n_i, v_j, n_j)$.

A variable v_i is observable at v_j if *eval(OTE)* is true and v_j is observable. Let *Obv(v,n)* be true if variable v is observable at control step n. The variable at which others are observed is usually a PO or a MISR. Hence, observability, *Obv(v,n),* of a variable v on the n^{th} control step is the ability to observe its value at a primary output or on-chip MISR.

Definition 3.11: Testability TE of a variable v (*vTTE*) in control step n with a controllability TE *vCTE* and observability TE *OTE* is the union defined by

$$vTTE(v,n) = vCTE(v,n) \bigcup OTE(v,n) \qquad (3.3)$$

An SDFG variable, v, in control step n is testable if there exists a testability TE (vTTE(v,n)) that evaluates to *True* such that all the STA constraints in it are compatible and *Obv(z,m)* is true, where z is a variable at which v is to be observed at a control step m. Therefore, the testability of an SDFG variable v, *vTest(v,n),* is defined as

$$vTest(v,n) = eval(vTTE(v,n)) \wedge Obv(z,m) \qquad (3.4)$$

Definition 3.12: Testability TE of an operation p with inputs x, y and output z is the union of its controllability TE and observability TE defined as

$$opTTE(p,n) = opCTE(p,n) \bigcup OTE(z,n+1) \qquad (3.5)$$

A 2-input SDFG operation, p, with inputs x and y, and output z is testable if there exists a testability TE (*opTTE(p,n)*) that evaluates to *True* such that all the STA constraints in it are compatible and *Obv(z,m)* is true, where z is a variable at which p is to be observed at a control step m. Therefore, the testability of an SDFG operation *p, opTest(p,n),* is defined as

$$opTest(p,n) = eval(opTTE(p,n)) \wedge Obv(z,m) \qquad (3.6)$$

The whole SDFG is testable if all the variables and all the operations are testable. Therefore, the testability of the whole SDFG, DTest, is defined as

$$DTest = \bigwedge_{i=1}^{NP} opTest(p_i, n_i) \wedge \bigwedge_{j=1}^{NV} vTest(v_j, n_j) \qquad (3.7)$$

The SDFG is, therefore, testable if the Boolean equation (3.7) evaluates to *True*. In the equation (3.7), p_i is any operation and n_i

is the control step at which it is scheduled, v_j is any variable and n_j is the control step at which it is scheduled, *NP* is the number of operations and *NV* is the number of variables in the SDFG. A symbol $\wedge$ implies the Boolean *AND* operation.

3.3 Test Environment Sharing

To illustrate the idea of test environment options, consider the SDFG in Figure 3.2. Inputs and outputs of the operations are variables, and the opTTEs of a given operation are used to test the associated functional module that performs that operation. To test, for example, a multiplier operation **3* (node **3*) using TPGs placed at the inputs of operations **1* and **2*, and a MISR at the output of *+4*, we need to set variables V_6 and V_7 to *gPSTAV* (control V_6 and V_7 to general controllability value) in the control step 2. This means that STA constraints $C_g(V_6,2)$ and $C_g(V_7,2)$ need to be satisfied to make test patterns reach the operation **3*. We also need to observe the test response value from V_8 in control step 3 (that is $Obv(V_8,3)$).

The respective variable controllability TEs for V_6 and V_7 are given by $vCTE(v_6,2)=\{C_g(V_6,2)\}$ and $vCTE(v_7,2)=\{C_g(V_7,2)\}$. Further derivation gives that

$vCTE(v_6,2)=\{C_g(V_1,1), C_1(V_2,1)\}$ or $vCTE(v_6,2)=\{C_1(V_1,1), C_g(V_2,1)\}$, *and*

$vCTE(v_7,2)=\{C_g(V_3,1), C_1(V_4,1)\}$ or $vCTE(v_7,2)=\{C_1(V_3,1), C_g(V_4,1)\}$.

To observe the value of V_8 in step *3* we derive its observability TE, which is given as $OTE(V_8,3)=\{C_0(V_5,1)\}$ and observe the test response, *rPSTAV*, at the variable V_9 in step *4*, i.e. $Obv(V_9,4)$.

By analyzing the functionality of the SDFG operations, it is observed that the variable V_6 has two alternative variable controllability TE options that can be used to control it to a value C_g in control step 2. Similarly, two options exist for achieving the controllability value $C_g(V_7,2)$ and one option for achieving the observability value $Obv(V_8,3)$. Further analysis shows that there are four (2 x 2 x 1) different alternative operation testability TE options for testing the operation **3* (see Table 3.1).

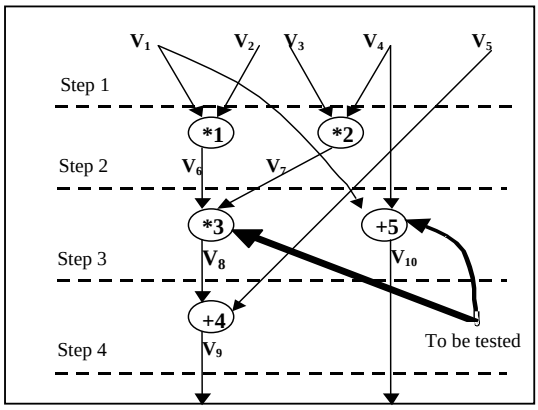


Figure 3.2 An SDFG example

Table 3.1 Alternative opTTEs for testing **3* and *+5*

Test Environments	**Constraints for controlling operations**	**Constraints for observing responses**
$opTTE_1$(*3,2)	$C_g(V_1,1), C_1(V_2,1), C_g(V_3,1), C_1(V_4,1)$	$C_0(V_5,1)$
*$opTTE_2$ (*3,2)*	*$C_g(V_1,1), C_1(V_2,1), C_1(V_3,1), C_g(V_4,1)$*	*$C_0(V_5,1)$*
$opTTE_3$(*3,2)	$C_1(V_1,1), C_g(V_2,1), C_g(V_3,1), C_1(V_4,1)$	$C_0(V_5,1)$
$opTTE_4$(*3,2)	$C_1(V_1,1), C_g(V_2,1), C_1(V_3,1), C_g(V_4,1)$	$C_0(V_5,1)$
$opTTE_1$(+5,2)	$C_g(V_1,1), C_g(V_4,1)$	-

To illustrate the idea of *sharing test environments*, let us consider operation *+5*. The testability TE for the operation *+5* (Figure 3.2) is given as *opTTE(+5,2)={$C_g(V_1,2)$, $C_g(V_4,2)$}* since for observing test responses, no constraints need to be satisfied. The test result has to be observed at the variable *V_{10}*. After simplifying *opTTE(+5,2),* we get, for the left input, $C_g(V_1,2) = C_g(V_1,1)$, for the right input, $C_g(V_4,2) = C_g(V_4,1)$ and for the observability of the output, $Obv(V_{10},3) = Obv(V_{10},4)$. Hence, *opTTE(+5,2)={$C_g(V_1,1)$, $C_g(V_4,1)$}*.

Suppose that the given SDFG variable, say v_k, needs to be set to $C_\alpha(v_k,\beta)$, where $\alpha \in STIPSTAV$, in the operation testability TEs (*opTTEs*) of different operations $OP=\{p_1, p_2, \ldots, p_n\}$, that is $C_\alpha(v_k,\beta) \in opTTE(p_1,s_1)$, $C_\alpha(v_k,\beta) \in opTTE(p_2, s_2)$, …, $C_\alpha(v_k,\beta) \in opTTE(p_n, s_n)$, where s_i is the control step at which the operation p_i is scheduled. Since the controllability value $C_\alpha(v_k,\beta)$ is common to the test environments of all the operations in the set *OP,* it can be shared by those operations to perform their concurrent testing. For example, consider variables V_1 and V_4 in the test environments of **3* and *+5* as discussed above.

As shown in Table 3.1, both the second test environment alternative of **3 ($opTTE_2$(*3,2))* and the test environment of *+5 ($opTTE_1$(+5,2))* need V_1 and V_4 to be controlled to C_g in the control step *1*, i.e. $C_g(V_1,1)$ and $C_g(V_4,1)$. Therefore, $C_g(V_1,1)$ and $C_g(V_4,1)$ can be shared to perform concurrent testing of both operations using the test environment $opTTE_2$ of the operation **3*.

3.4 Alternative Test Environment Options

As discussed in the previous section, there, possibly, exist more than one test environment for controlling input operands and observing test responses for each operation in the SDFG. To explain this idea, the following definitions are provided:

Definition 3.13: ${}_{alt}C_0(v_i,n)$ is defined as the number of compatible alternative test environment options (ATEO) that can be used to set variable v_i to a controllability value *0PSTAV* in control step *n*. □

Definition 3.14: ${}_{alt}C_1(v_i,n)$ is defined as the number of compatible ATEOs that can be used to set variable v_i to a controllability value *1PSTAV* in control step *n*. □

Definition 3.15: ${}_{alt}C_g(v_i,n)$ is defined as the number of compatible ATEOs that can be used to set variable v_i to a controllability value *gPSTAV* in control step *n*. □

Definition 3.16: ${}_{alt}O(v_i,n)$ is defined as the number of compatible ATEOs that can be used to enable observability of a variable v_i in control step *n* at some signature registers. □

Two test environment alternatives are compatible if and only if every variable that is included in both of them needs to be controlled to the same value and at the same control step. However, two ATEOs need not necessarily have exactly the same number and type of variables. They can have some different variables, but the common ones have to be consistent.

If we want to observe the node N_1 in Figure 3.3, we need to observe variable v_{tbo} (v_{tbo} and v_{tbc} stand for a variable to be observed and a variable to be constrained to controllability value *0PSTAV*, respectively). Based on STA, this implies constraining v_{tbc} to *0PSTAV*, and observing the value of v_{tbo} at any of the observable output variables (v_{o1}, v_{o2}… v_{on}) at the output of node N_3. Therefore, the number of observability alternatives increases when the node N_3 has multiple observability paths, which, in this case, are also inherited by the node N_1, provided that v_{tbc} can be constrained to *0PSTAV*.

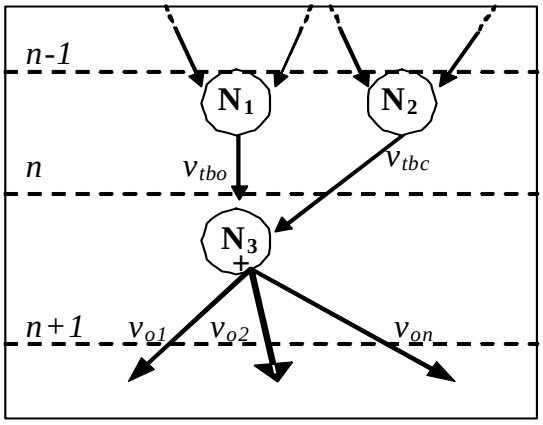


Figure 3.3 Multiple alternative observability paths

In Figure 3.3, the number of observability alternative options for the variable $v_{tbo,}$ denoted as ${}_{alt}O(v_{tbo},n)$ is ${}_{alt}C_0(v_{tbc},s_0)\times{}_{alt}O(v_{o1},s_1)+{}_{alt}C_0(v_{tbc},s_0)\times{}_{alt}O(v_{o2},s_2)+\ldots+{}_{alt}C_0(v_{tbc},s_0)\times{}_{alt}O(v_{on},s_n)$. This leads to the equation (3.8).

$$_{alt}O(v_{tbo},n)=_{alt}C_0(v_{tbc},s_0)\times\sum_{i=1}^{n}{}_{alt}O(v_{oi},s_i) \qquad (3.8)$$

3.5 Testability of RTL Modules and Registers

If an RTL design is synthesized from an SDFG representation, a given RTL functional module can implement a number of operations. Similarly, a number of variables can be implemented by the same RTL register. This thesis presents a post-HLS testability enhancement methodology, which is applied to RTL designs. It is applied after scheduling, allocation and binding are done. The essence of our approach is the fact that high-level behavioral information is used for testability analysis. The approach also makes use of the allocation/binding information provided by the HLS step.

From the discussion in Section 3.4 we know that each variable v in the SDFG has a set of compatible ATEOs (for each controllability value in the set *STIPSTAV*). Suppose that the set of ATEOs for controlling variable v to *gPSTAV* is given as $_{g}A(v,n)$. Similarly, $_{0}A(v,n)$, $_{1}A(v,n)$, and $_{a1}A(v,n)$ are the respective sets of ATEOs for controlling the variable v to *0PSTAV*, *1PSTAV* and *a1PSTAV*. $_{obv}A(v,n)$ and $_{t}A(v,n)$ are the set of ATEOs for observing and testing variable v respectively.

Suppose that a set of SDFG variables $SV=\{v_1, v_2 \ldots v_v\}$ is mapped to a register *r*. Since each variable in the set *SV* is mapped to the same physical register, it is sufficient to test only one of the variables mapped to it. Testability of the register *r* is therefore expressed by the Boolean summation of the testabilities of all the variables it implements, as shown in equation (3.9).

$$Test(r)=\bigvee_{i=1}^{|SV|}vTest(v_i,s_i), \qquad (3.9)$$

where $v_i \in SV$ and s_i is the control step at which the variable v_i is scheduled.

Since a variable can possess a number of ATEOs that can be used to test it, the testability of the register *r* can be further expanded by using these ATEOs as shown in equation (3.10).

$$Test(r) = \bigvee_{i=1}^{|SV|} (a_{i,1} \vee a_{i,2} \vee ... \vee a_{i,|_t A(v_i)|}) = \bigvee_{i=1}^{|SV|} \bigvee_{j=1}^{|_t A(v_i)|} a_{i,j}, \qquad (3.10)$$

where $a_{i,j}$ denotes the j^{th} testability ATEO of variable v_i and $|_tA(v_i)|$ is the number of ATEOs that can be used to test variable v_i.

To define testability of the RTL functional modules, an argument similar to that used in defining testability of registers is used. Suppose a set of SDFG operations $SO=\{p_1, p_2 ... p_n\}$ is mapped to a functional module *m*. Since all operations in *SO* are mapped to the same functional module *m*, testing any one of them is sufficient to test the module. Testability of the module *m* is, therefore, expressed by the equation (3.11), which leads to the equation (3.12) upon expansion using the testability ATEOs.

$$Test(m) = \bigvee_{i=1}^{|SO|} opTest(p_i, s_i), \qquad (3.11)$$

where $p_i \in SO$ and s_i is the control step at which the operation p_i is scheduled.

$$Test(m) = \bigvee_{i=1}^{|SO|} (a_{i,1} \vee a_{i,2} \vee ... \vee a_{i,|_t A(p_i)|}) = \bigvee_{i=1}^{|SO|} \bigvee_{j=1}^{|_t A(p_i)|} a_{i,j}, \qquad (3.12)$$

where $a_{i,j}$ denotes the j^{th} testability ATEO for the operation p_i and $|_tA(p_i)|$ is the number of existing ATEOs that can be used to test the operation p_i.

The whole RTL design, *DRT*, consisting of a set *SM* of functional modules and a set *SR* of registers is testable if all the registers and all the functional modules are testable. Thus the RTL design is testable if the Boolean equation (3.13) evaluates to *True*.

$$Test(DRT) = \bigwedge_{i=1}^{|SM|} Test(m_i) \wedge \bigwedge_{j=1}^{|SR|} Test(r_j), \qquad (3.13)$$

where $m_i \in SM$ and $r_j \in SR$.

3.6 Structure of the Rest of the Thesis

The rest of the thesis describes the contributions in detail.

Chapter 4 deals with the minimization of BIST resources under testing time constraints. To minimize BIST resources and schedule testing of the operations, we explore the parallelism inherited from the nature of the design. This means that the topology of the design representation is analyzed. Using alternative test environment options captures degree of sharing of the BIST resources. Therefore, by exploring sharing of the test environments we try to minimize BIST resources. We assume that the designer imposes constraints on the testing time. The BIST resources are optimized such that the testing time constraints and full testability are satisfied. In Chapter 4 we assume a classical approach whereby the BIST resources optimization strategy simply uses the cost of functional, BIST and multiplexer hardware modules as an optimization objective.

Chapter 5 extends the BIST resources optimization problem with wiring consideration and proposes two optimization approaches. This problem is motivated by the fact that the total area of the design is not only composed of the area of the BIST and functional modules, but also wiring area. In deep sub-micron technology wiring constitutes a substantial amount of chip area. Therefore, in this chapter we study the problem of optimizing the total area of the design, including wiring, while ensuring that each module is testable.

Chapter 4
Testing-Time Constrained BIST Synthesis

This chapter describes an approach to solve the problem of optimizing BIST resource usage under full-testability and testing time constraints described in Chapter 1. The test-problem identification and BIST enhancement strategy during the optimization process are assisted by symbolic testability analysis. Since the problem we address is NP hard, we have developed heuristics to solve it.

4.1 BIST Synthesis Overview

Our approach first analyzes and improves the testability of the design. After that it determines the initial testing time, T_{init}, which can be achieved as a result of the parallelism, inherited from the nature of the design itself. In simple terms, T_{init} is the testing time needed to test the design after using our heuristic to achieve one hundred percent testability of the design.

We have defined a test session as a group of modules that are tested concurrently and a test schedule as a set of all test sessions for testing a given design.

We have assumed that pseudo-random BIST technique will be used to test the design and that the same number of pseudo-random test vectors will be used for testing each functional module. Thus, all test sessions will be of equal length, and the

total testing time will be directly proportional to the number of test sessions. Consequently, T_{init} is the number of test sessions needed to test the design after using our heuristic to achieve one hundred percent testability of the design multiplied by the length of the test session.

Detailed explanation on how T_{init} is obtained is provided in sub-sections 4.2 through 4.5.

Given a certain required testing time constraint, T_{req}, the following alternatives are taken:

- If $T_{req}<T_{init}$, shrink the test schedule by adding a minimal amount of hardware such that the test time constraints are satisfied;
- If $T_{req}=T_{init}$, optimize BIST hardware, so that minimal overhead is left;
- If $T_{req}>T_{init}$, optimize the BIST hardware by stretching the test schedule, such that minimal overhead is left and testing time is $T_{BIST} \leq T_{req}$ (T_{BIST} approaches T_{req}).

In summary, our BIST time analysis and resource optimization approach works with RTL designs represented in a notation based on SDFG. The outputs are: a test schedule that satisfies testing time constraints and an RTL design with minimal added BIST resources.

Our overall BIST testing time analysis and resources optimization approach is described in Figure 4.1. STA is used to select untestable operations and testability enhancement is performed. In the course of STA all possible testability test environments for each operation are also extracted. Then, sets of operations that cannot be tested concurrently due to MISR sharing conflicts are identified. Test session selection heuristic is used to select concurrent test sessions based on test environment options (see Section 3.3 and 3.4) and MISR sharing conflicts. The objective is to minimize the length of the test schedule. The testing time of the resulting test schedule is denoted as T_{init}. After that, the time constraints are considered and appropriate steps taken as discussed previously in this section. Finally, a merged design and BIST controller, and a BIST-ed data path are generated.

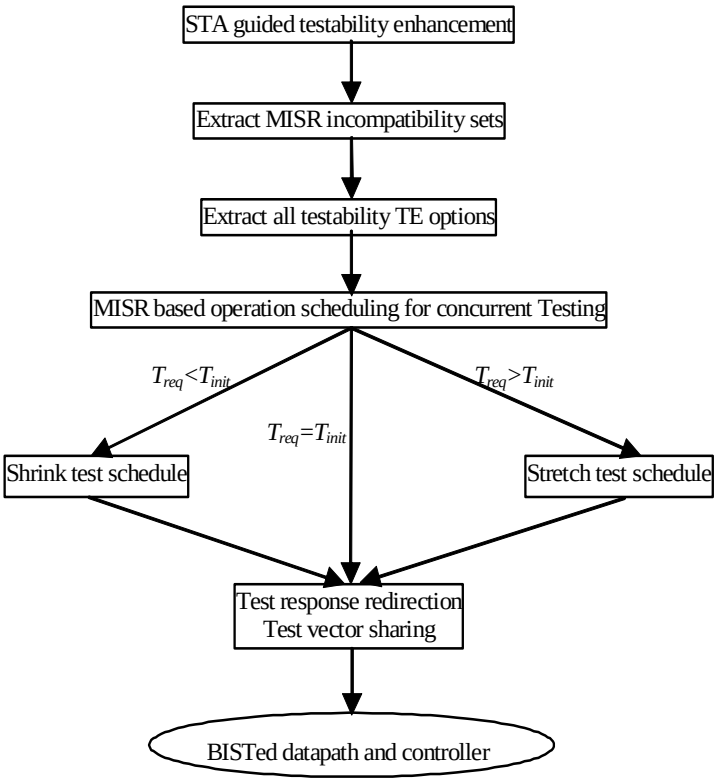


Figure 4.1 Overview of BIST resources optimization strategy

4.2 Testability Enhancement

The basic idea behind our testability enhancement is a conversion of functional registers to BIST registers or the insertion of dedicated BIST registers. We assume that these BIST registers have dual modes, functional and testing. Consequently, they can be configured as storage units during functional mode or as on-chip test pattern generators and/or signature analyzers during test mode.

In this Chapter we propose a testability enhancement heuristic that aims at adding a small amount of BIST resources that will guarantee 100% testability for all the modules in the design. The heuristic does not guarantee that the added BIST hardware is optimal. The testability enhancement is performed sequentially in three steps. In the first step controllability enhancement to 100% is performed, then in the second step observability is enhanced to 100% and finally global testability is enhanced to 100%.

It can be observed that, very often, uncontrollable nodes induce controllability problems to all successor nodes. This is caused by data dependency resulting from the topology of the SDFG. Our controllability enhancement strategy, thus, first enhances the node that is the source of controllability problems. Consequently, enhancing one node can improve controllability of most of the successor nodes.

Our controllability enhancement algorithm is depicted in Figure 4.2. It starts by analyzing testability of the SDFG to find all uncontrollable operations (line 1). Then the process of controllability enhancement is repeated until all operations become controllable (lines 2-22). Each time when there are still some uncontrollable operations left, we group uncontrollable operations into connected groups. These connected operations are referred to as uncontrollable sub-graphs (UCSG) (line 3). The largest group is referred to as largest uncontrollable sub-graph (LUCSG) (line 4). The idea is to enhance controllability of an operation that is at the top (TOP) of the LUCSG (line 5) so that its controllability can be propagated to other operations in the group.

By first deciding which input register of the operation is to be enhanced and then converting it to a TPG if it is a primary input or a BILBO otherwise achieve controllability enhancement. If the operation has only one uncontrollable input then this input is enhanced. If both inputs of the operation are uncontrollable, we prioritize enhancing controllability of the left input register.

Algorithm: EnhanceControllability

```
Begin
1.  (Uncontrollable, Unobservable, Untestable) ← STA(SDFG);
2.  while (Uncontrollable != ϕ) do
3.      UCSG ← getUncontrollableSubGraphs(Uncontrollable);
4.      LUCSG ← getLargestUncontrollableSubGraph(UCSG);
5.      TOP ← get operation at the top of LUCSG;
6.      X ← getInput(left, TOP);
7.      Y ← getInput(right, TOP);
8.      if X not controllable then
9.          if X is a PI then
10.             modifyDesign(SDFG, X, TPG);     //convert X to TPG
11.         else
12.             modifyDesign(SDFG, X, BILBO);  //convert X to BILBO
13.         end if
14.     else
15.         if Y is a PI then
16.             modifyDesign(SDFG, Y, TPG);     //convert Y to TPG
17.         else
18.             modifyDesign(SDFG, Y, BILBO);  //convert Y to BILBO
19.         end if
20.     end if
21.     (Uncontrollable, Unobservable, Untestable) ← STA(SDFG);
22. end while
End.
```

Figure 4.2 Controllability enhancement algorithm

Unobservable modules are usually buried far from POs or MISRs. Observability of a module imposes restrictions on the values of other variables in order for the test responses to be propagated to the MISRs. Sometimes the restrictions are not able to force propagation of the values to MISRs and in some cases some variables are simultaneously forced to have contradictory values to enable observability, thus, these operations become unobservable as shown in Figure 4.3.

If node N_1 in Figure 4.3 is to be tested, controllability value $C_g(V_1,1)$ is to be set at V_1 and $C_g(V_2,1)$ at V_2 while the output of N_2 has to be controlled to $C_0(V_5,2)$ to enable observability of the output of N_1 at a MISR. Since V_2 is also connected to N_2, whatever value is set at V_3, $C_0(V_5,2)$ cannot be guaranteed at the output of N_2, hence, test responses at the output of N_1 cannot reach the MISR.

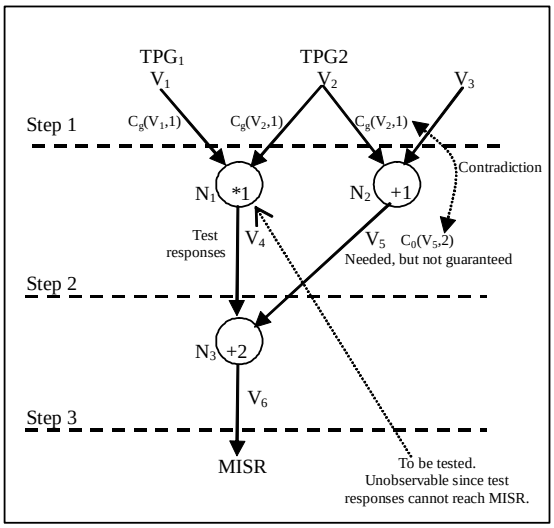


Figure 4.3 Observability problem due to contradictory values on intermediate nodes

One solution to the observability problem discussed above is to introduce a MISR at the output of the node N_1 or redirect test responses from N_1 to an existing MISR in the design. However, in more complex designs, this has to be done in a way such that MISR resources are efficiently used. Therefore, our BIST observability enhancement heuristic is to add a dedicated MISR at the output of a node situated at the end of a sub-graph of unobservable nodes. If a MISR is added to improve an unobservable node that is not at the end of the unobservable sub-graph, then the downstream modules will still be unobservable. This idea is illustrated in Figure 4.4. Before BIST enhancement, the design has three primary input variables (V_1, V_2 and V_3) and three constant nodes (C_1, C_2 and C_3). STA reveals the existence of two unobservable sub-graphs. The first one consists of nodes **1*, **2** and **3* whereas the second consists of **2, *4* and *–1*. To enhance the observability of these sub-graphs, our approach selects to enhance the observability of variables V_9 and V_{10}, which are at the end of the first and second unobservable sub-graphs respectively. As a result, the observability of all three nodes in each of the two sub-graphs is enhanced. Had we, for example,

enhanced observability of variable *V5* instead, only observability of node **2* would have been enhanced. Consequently, it would have been necessary to add more MISRs to improve the observability of the remaining four nodes. Therefore, our approach selects places to enhance observability such that the smallest number of MISRs and BILBOs is added into the design as shown in Figure 4.4.

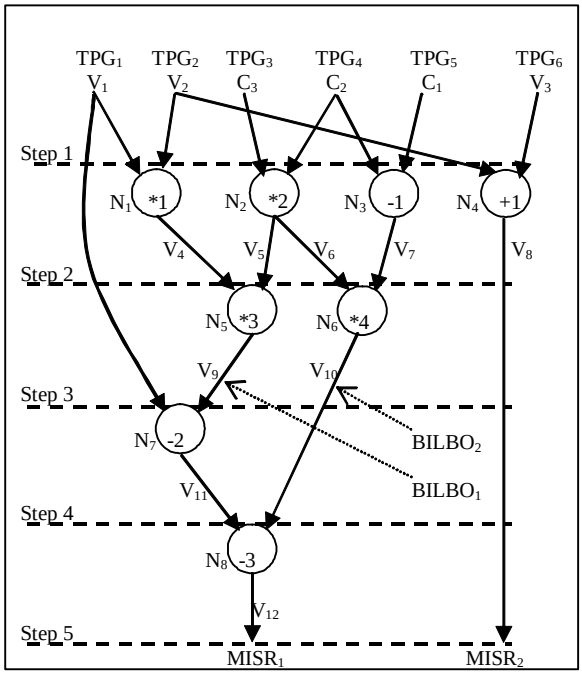


Figure 4.4 Selecting observability enhancement places

Our complete observability enhancement algorithm is depicted in Figure 4.5. It starts by running STA to find all unobservable operations (line 1). Then the process of observability enhancement is repeated until all operations become observable (lines 2-15). Each time when there are still some unobservable operations, we group them into unobservable sub-graphs (UOSG) as discussed in the previous paragraph. Then the largest unobservable sub-graph, LUOSG, is found (line 4). The idea is to enhance observability of an operation (BOP) that is at the bottom of the LUOSG (line 6) so

that other unobservable operations up the sub-graph will also be enhanced. For the chosen operation, we enhance observability of one of its output register, which is converted to a MISR if it is a primary output, otherwise to a BILBO.

Algorithm: EnhanceObservability

```
Begin
  1.  (Uncontrollable, Unobservable, Untestable) ← STA(SDFG);
  2.  while (Unobservable != ϕ) do
  3.      UOSG ← getUnobservableSubGraphs(Unobservable);
  4.      LUOSG ← getLargestUnobservableSubGraph(UOSG);
  5.      // EnhanceObservability(LUOSG);
  6.      BOP ← get operation at the bottom of LUOSG;
  7.      Z ← getOutputRegisters(BOP);
  8.      if R∈Z is a PO then
  9.          convert R to MISR;
  10.     else
  11.         R ← Z[0];
  12.         convert R to BILBO;
  13.     end if
  14.     (Uncontrollable, Unobservable,Untestable) ← STA(SDFG);
  15. end while
End.
```

Figure 4.5 Observability enhancement algorithm

After controllability and observability are enhanced, it is still possible that the design will not be testable. Therefore, testability of the design has to be re-checked, and if there are still some untestable modules, then their testability has to be enhanced.

Note that a module is considered testable if it is simultaneously controllable and observable. It is, therefore, possible that a module that is both controllable and observable can still be untestable. This can happen if a given SDFG variable is required to be set to different values at the same time, one for enabling controllability and another for enabling observability. Consequently, the associated module becomes untestable since two different values cannot be set to the same variable at the same time [50]. To solve the problem, we have proposed a testability enhancement algorithm shown in Figure 4.6. It is based on the same idea as the

controllability and the observability algorithms. The untestable operations are grouped in sub-graphs and the largest one is improved first (line 4). The operation at the bottom of the largest untestable sub-graph is prioritized for enhancement. Since an operation usually has two inputs and one output, the priority of enhancement is in the order left input, right input and then output. Since the controllability and the observability enhancements have been done, all PIs and POs have been enhanced for BIST. Therefore, testability enhancement mainly targets internal registers, which are usually converted into BILBOs. The algorithm proceeds as shown in Figure 4.6.

Algorithm: EnhanceTestability

```
Begin
 1.  (Uncontrollable, Unobservable, Untestable) ← STA(SDFG);
 2.  while (Untestable != ϕ) do
 3.      UTSG ← getUntestableSubGraphs(Untestable);
 4.      LUTSG ← getLargestUntestableSubGraph(UTSG);
 5.      // Enhance testability
 6.      BOP ← get operation at the bottom of LUTSG;
 7.      X ← getInput(left, BOP);
 8.      Y ← getInput(right, BOP);
 9.      Z ← getOutputRegisters(BOP);
10.      // Enhance controllability
11.      if X != BILBO and hasInputs(X) then
12.          convert X to BILBO
13.          goto checkTestability;
14.      if Y != BILBO and hasInputs(Y) then
15.          convert Y to BILBO
16.          goto checkTestability;
17.      // Enhance observability
18.      if R∈Z is a PO then
19.          convert R to MISR;
20.      else
21.          R ← Z[0];
22.          convert R to BILBO;
23.      end if
24.      checkTestability:
25.          (Uncontrollable, Unobservable, Untestable)← STA(SDFG);
26.  end while
End.
```

Figure 4.6 Testability enhancement algorithm

4.3 MISR Sharing

Simultaneous analysis of test responses from multiple functional modules requires the availability of as many MISRs as there are modules that are to be analyzed at the same time. This means that the number of available MISRs bounds the number of modules, which can be simultaneously analyzed. In addition, all concurrently analyzed modules must have all their inputs simultaneously controlled by setting appropriate controllability values on the variables as given in their testability TEs. Modules that are analyzed using the same MISR must be scheduled in different test sessions due to MISR sharing conflicts.

4.4 MISR Incompatibility Sets

MISR incompatibility sets (MISRISs) consist of operations that cannot be concurrently tested due to MISR sharing conflicts. Two operations are contained in the same set if they share the same MISR for test response analysis and, therefore, cannot be concurrently tested.

STA results give sufficient information for extracting MISRISs. To extract MISRISs we group operations based on the signature registers that are used to analyze their responses. Each signature analysis register, M_i, corresponds to one set, G_i, which will include all operations that are analyzed by it. All operations in the same set are known as incompatible operations with respect to their corresponding MISR.

The number of incompatible operations in the largest MISRIS determines a lower bound on the minimal number of test sessions that are needed for testing the whole design. In reality, the total testing time is not only determined by MISR sharing incompatibilities, but also is constrained by the choice of good test environment options, which determine whether the *TEs* are conflict free to enable concurrent testing of the modules.

4.5 Concurrent Test Session Selection

Once the MISRISs are available, the next step is to select concurrent test sessions. A group consisting of one operation from each MISRIS can possibly be tested concurrently if the operations will not violate the test environment constraints.

If the test environment constraints are not considered, it can be possible to schedule operations in a minimal number of test sessions equal to the maximum number of operations in the most congested MISRIS. However, these may not be correct test sessions because the availability of MISRs for concurrent observation of responses does not guarantee that those operations can be properly controlled and the responses properly propagated to the corresponding MISR registers at the same time for all tested operations in a given test session. In this way, controllability constraints imposed by the test environments of individual operations may cause an increased number of test sessions. This is due to the fact that there may exist operations that use different MISRs for signature analysis, but compete for the same variables to control their inputs or propagate test response to the appropriate MISR, hence cannot be simultaneously controlled.

Test environments have two components. The first component consists of the controllability values necessary to control the inputs of the operations and the second component consists of the controllability values necessary to force propagation of test responses to the corresponding MISR. Thus, when constraints due to both controllability of the input operands and those imposed to propagate test responses to the appropriate MISR are taken into account during the test session selection process, an increase in the number of test sessions will be noticed and the MISRs will be less effectively used, with some of them remaining idle during several test sessions. After all constraints are taken into consideration, the resulting number of test sessions represents the initial testing time, T_{init}. Thus, it is possible to test the design in T_{init} test sessions as a result of the nature of parallelism inherited from the design itself.

Out of the compatible test environment alternatives, the particular testability TE alternative option that minimizes MISR conflicts and

can lead to packing as many operations as possible in each test session will be chosen. Consequently, the total number of test sessions will be minimized. In addition, *TE*s of all operations in a test session must be simultaneously supported. When the best choice of *TE* alternatives is achieved, the associated testing time is the initial testing time, T_{init}. The best choice among the alternative *TE* options is the one targeted to favor maximum parallelism in testing operations.

Our heuristic for selection of concurrent test sessions is based on an equal length test-scheduling algorithm [16]. We extended the algorithm to take into consideration controllability and observability constraints when choosing operations to be included in a given test session. Therefore, operations are included in the same concurrent test session not only if they do not share MISR, but also if controllability and observability constraints are satisfied for all of them at the same time.

4.6 BIST Resources Optimization

After T_{init} is obtained, further hardware optimization is performed until the requested test time constraint, T_{req}, is satisfied. Three different optimization cases are considered based on comparison of T_{init} and the test time constraint, T_{req}. In all cases the optimization objective is the cost of BIST registers and multiplexers as shown in equation (4.1), where b_i is the area of the i^{th} BIST register, B is the number of BIST registers, m_j is the area of the j^{th} multiplexer, and M is the number of multiplexers.

$$Cost = \sum_{i=1}^{B} b_i + \sum_{j=1}^{M} m_j \qquad (4.1)$$

Case 1: $T_{req} < T_{init}$*:*If the requested number of test sessions, T_{req}, is less than T_{init}, our approach optimizes hardware by shrinking the test schedule. Since we want to satisfy the constraint T_{req}, and up to this moment the hardware has been optimized in such a way that the design can be tested in T_{init} test sessions, our approach continues to shrink the test schedule by adding more hardware until the test time T equals T_{req}.

Hardware optimization by test schedule shrinking is described by the algorithm in Figure 4.7 and proceeds as follows: All T_{init} test sessions are ranked in decreasing order of the number of operations to be tested in that session (line 1). Suppose *rTS* is a set of test sessions ranked in decreasing order of the number of operations. This means that the first test session in *rTS* has the greatest number of operations. We take the first T_{req} ($T_{req} \leq T_{init}$) test sessions and make them default test sessions of our shrinked test schedule (STS). All the operations in the remaining test sessions are then considered unscheduled (lines 2-6) and will be re-scheduled in one of the test session $TS_i \in STS$, $1 \leq i \leq |STS|$. An operation will be placed in the test session in which it needs minimal additional hardware and can be concurrently tested with all the other operations in the same test session. Lines 7-21 of our shrink test schedule algorithm achieve this.

For each unscheduled operation, the set of all potential possible

```
Algorithm: ShrinkTestSchedule
Begin
 1. rTS ← rank test sessions in decreasing order of complexity;
 2. Uns ← ϕ ;      // unscheduled operations
 3. for i ← Treq+1, Treq+2, .., Tinit  do
 4.     Uns ← Uns ∪ rTS[i];
 5.     rTS ← rTS - rTS[i];
 6. end for
 7. STS ← rTS;   // initialize shrinked test schedule
 8. while (Uns ≠ ϕ) do
 9.     op ← get one unscheduled operation from Uns;
10.     Uns ← Uns – { op };
11.     Ψp ← compute potential TEs for op;
12.     for i ←1, 2, .., |STS| do
13.         TSi ← STS[i];      // getTestSession(i);
14.         Ω ← get TEs used in TSi;
15.         Ψc ← computeCompatibility(Ψp, Ω);
16.         bestATEO[i] ←getBestATEO for TSi from Ψc; //cheapest ATEO
17.     end for
18.     chosenATEO← a, a∈bestATEO | cost(a)=Min{bestATEO[i]}, 1≤i≤Treq;
19.     Schedule op in the test session where chosenATEO is;
20.     Add hardware to accommodate op in the chosen test session;
21. end while
22. Apply TestResponseRedirection algorithm;
End.
```

Figure 4.7 BIST optimization by test schedule shrinking

TEs, Ψ_p, is computed (line 11). During the computation process of the potential *TEs*, the STA procedure assumes that the internal non-BIST registers can be converted into BIST registers to provide more possibilities of the *TEs*. For each test session in the shrinked test schedule $TS_i \in STS$, $1 \leq i \leq |STS|$, we find all $TE \in \Psi_p$ that are compatible with all the operations in that session (lines 13-15). This process tries to find all the *TEs* for the given operation that can be used to schedule it in a given, existing test session. These compatible *TEs* are denoted as Ψ_c. For a given test session, the compatible $TE \in \Psi_c$ that will need the cheapest modification of the design is considered as the best candidate *TE* for scheduling the operation in that test session (line 16). The process is repeated to find best candidate *TE* for each test session for the given operation. Out of all candidate *TEs*, the candidate *TE* that needs the overall minimum cost is chosen and the operation is scheduled in the corresponding test session using the *TE* that incur the cheapest hardware cost. The design is then modified so that the chosen *TE* can be provided.

Case 2: $T_{req} = T_{init}$*:* Our approach explores the possibility of further hardware optimization by using the strategy of test response redirection. As it has been emphasized in the discussion in Section 4.5, several MISRs are not effectively used in some test sessions; hence, our approach recovers some of them and converts them back to normal registers. The operations that use recovered MISRs are redirected to other free MISRs in the same test session.

To optimize resource usage, one basic idea is to redirect test responses from some operations to other MISRs different from those originally assigned to, if the time constraints are not violated and the MISRs allow the redirection.

Let L_u represents a MISR that is least used in all test sessions. This means that L_u remains idle in most of the test sessions as compared to other MISRs. Let U be a set consisting of test sessions in which L_u is used. During execution of the algorithm, M_c is the set of currently used MISRs. When a MISR is recovered and converted back to a normal register, it is removed from M_c. F is a set consisting of MISRs that are free in every test session in which L_u is used. Among the free MISRs in set F, P is the one that is mostly packed, which means, P analyzes responses from the

greatest number of operations as compared to the other MISRs in *F*. Let *G* be the set of all MISR incompatibility sets. Given a certain MISR *X*, G_X represents the incompatibility set corresponding to MISR *X*. The algorithm shown in Figure 4.8 minimizes the hardware cost (see equation (4.1)) and produces the set M_C of used MISRs and the corresponding incompatibility sets. This optimization is performed without increasing the number of test sessions.

Algorithm: TestResponseRedirection

```
Begin
 1.  G ← set of all incompatibility sets;
 2.  Best_selection_obtained ← FALSE;
 3.  while (best_selection_obtained != TRUE) do
 4.      Lu ← X, X ∈ MC and X is least used;
 5.      U ← All test sessions in which Lu is used;
 6.      F ← Free MISRs in sessions U;
 7.      If F ≠ϕ then
 8.          P ← X, X ∈ F and X is most packed;
 9.          for every operation op∈GLu do
10.              connect operation op to MISR P;
11.          end for
12.          if cost is reduced then
13.              G ← G – { GP, GLu };
14.              GP ← GP U GLu;
15.              G ← G U { GP };
16.              MC ← MC - { Lu };
17.          else
18.               Discard all connections done in line 10;
19.          end if
20.      else
21.          best_selection_obtained ← TRUE;
22.      end if
23. end while
24. return MC, G;
End.
```

Figure 4.8 BIST optimization by test responses redirection

Case 3: $T_{req} > T_{init}$*:* As discussed above, initially all operations are scheduled in T_{init} test sessions, which are denoted as TS_i, $1 \le i \le T_{init}$. If $T_{req} > T_{init}$, our approach increases the testing time from T_{init} to T_{BIST} by stretching the test schedule in such a way that the required time constraint is satisfied ($T_{init} < T_{BIST} \le T_{req}$). In this case, we can recover more BIST hardware resources that may not necessarily be needed. The success of this optimization depends on test environment conflicts of the operations and on how large T_{req} is compared to T_{init}.

To optimize BIST hardware by stretching the test schedule, first we find two most utilized TPGs and the most utilized MISR. Since these resources are already highly utilized, we will keep them. Then we will try to recover as many of the remaining BIST registers as test time constraint will allow. These remaining BIST registers are referred to as candidate registers for recovery and are denoted as *Candid*.

First of all, the remaining BIST registers are ranked in increasing degree of utilization, i.e. the least used register first. These remaining BIST registers are candidates for removal in order to minimize hardware while satisfying given a testing-time constraint. Our hardware reduction technique utilizes the longer testing-time freedom to reduce hardware overhead.

To remove a candidate BIST register $b \in Candid$, we find all operations which use b for BIST activities, and then we search for alternative connections (cheapest ones) to perform BIST activities for those operations. If these connection alternatives exist and they make the design testable and if the operations can be scheduled in other test sessions and they lead to hardware reduction, then b is removed (converted to functional register). Otherwise the BIST register b and test schedule are left intact. This process is repeated for all candidate registers. The detailed stretch test schedule algorithm is described in Figure 4.9.

Algorithm: TestScheduleStretching

```
Begin
1.  PrTPG ← get 2 most utilized TPGs;
2.  PrMISR ← get one most utilized MISR;
3.  BR ← set of BIST resources ranked in increasing order of utilization;
4.  BR ← BR – {PrTPG} – {PrMISR};
5.  STS ← get test schedule of size T_init; // STS is stretched test schedule
6.  ESTS ← extend STS with empty test sessions till |ESTS|=T_req;
7.  RC ← ϕ;  // recovered BIST registers;
8.  for i ← 1, 2, .., |BR| do
9.      b ← BR[i];
10.     BR ← BR – {b};
11.     OP ← all operations which use b for BIST activities;
12.     D ← find alternative BIST for OP by connections;
13.     ConnectAlternativeBIST(OP, D);
14.     Testable ← Check testability by doing STA;
15.     Schedulable ← ScheduleOperations(OP, ESTS);
16.     if (Schedulable and Testable and cost is reduced) then
17.         Convert b to normal register;
18.         RC ← RC U { b };
19.         ESTS ← UpdateSchedule(OP);
20.         Accept design modification;
21.     else
22.         Cancel design modification;
23.         BR ← BR U { b };
24.     end if
25. end for
26. apply TestResponseRedirection algorithm;
27. Return RC, TSS;
End.
```

Figure 4.9 BIST optimization by test schedule stretching

The other idea to reduce BIST overhead is to share test pattern generators among operations. In order to effectively share TPGs, our approach first chooses two most utilized TPGs and assumes that they cannot be removed. For each of the remaining TPGs, we find all operations, which get test patterns from it and connect them to other alternative TPGs in the design. Then the TPG is removed. If the design becomes testable and operations schedulable in the same number of test sessions and the BIST overhead is reduced then the TPG is permanently removed. Otherwise the TPG is put back in the design.

Additional multiplexers and wiring will be needed in order to redirect test responses for analysis to different MISRs or to share test patterns among operations. After some TPGs and MISRs are disabled as BIST registers, they will still remain in the design as normal registers for their functional storage use, hence not adding any BIST overhead.

4.7 Experimental Results

Researchers on high-level BIST insertion typically evaluate the efficiency of their approaches by comparing the amount of BIST hardware added. This is usually computed as the number of TPGs, MISRs, BILBOs and CBILBOs added. On the other hand, our approach takes the area of the BIST registers and multiplexers as the optimization objective.

Sizes of the functional registers and modules are adopted from [51]. For the BIST registers, we have assumed a simple relationship between the sizes of the functional and BIST registers: Register < TPG < MISR < BILBO < CBILBO. The areas of the 16-bit modules used in the experiments are shown in Table 4.1.

Table 4.1 Sizes of modules and registers

Name	**Area (μm^2)**
Adder	50,000
Subtractor	50,000
Multiplier	250,000
Divider	250,000
Register	15,000
TPG	20,000
MISR	30,000
BILBO	40,000
CBILBO	50,000
Multiplexer	$1000 + 500 \times N$

, where N is the number of multiplexer inputs

We have tested our approach on several HLS benchmarks. Characteristics of the designs used in our experiments are summarized in Table 4.2. The designs have been synthesized using a very simple HLS algorithm such that each SDFG operation

is implemented using a separate functional module. Details of the design features can be found in [44].

Table 4.2 Characteristics of the designs

Design name	#Adders	#Subtractors	#Multipliers	#Dividers	#LogicAND
Tseng	3	1	2	1	1
Real	3	2	4	2	0
Paulin	2	2	6	0	0
Overnctrl	5	1	1	1	0
Ewf	26	0	8	0	0

The first set of experimental results (Table 4.3, Table 4.4) deals with our initial testability enhancements and test hardware optimization by test response redirection and test patterns sharing. In these experiments, testability degree is computed as a percentage of fully testable operations. An operation is testable if all its input operands are controllable and its output observable at the same time. Controllability degree is the percentage of the operations that are controllable, that is, the ratio of the controllable operations to the total number of operations in the design times one hundred percent. In order for the operation to be counted as controllable, both its left and right hand operands must be simultaneously controllable. If any input operand is not fully controllable, the associated operation is assumed to be not controllable. Similarly, observability degree is the percentage of the observable operations in the design.

Table 4.3 Testability analysis results of the original designs

Design name	#TPG	#MISR	Controllabilitý degree	Observability degree	Testability degree
Paulin	4	3	40.0	80.0	30.0
Real	3	2	9.0	16.0	0.0
Overnctrl	5	2	75.0	62.5	37.5
Tseng	5	2	87.5	75.0	12.5
EWF	8	7	20.6	88.0	5.9

Table 4.4 BIST resources after testability enhancement and optimization to 100% testability

		Straightforward			Optimized			%Hardware reduction	
Name	**T_{init}**	**#TPG**	**#MISR**	**#BILBO**	**#TPG**	**#MISR**	**#BILBO**	**HW cost**	**Number of BIST regs**
Paulin	6	5	3	2	4	3	2	6.0	10.0
Real	3	6	2	5	6	2	3	19.0	15.4
Overnctrl	5	6	2	1	5	2	1	8.2	11.1
Tseng	4	5	2	3	5	2	2	12.9	10.0
EWF	7	9	7	12	11	5	5	31.0	25.0
Average								**15.4**	**14.3**

Table 4.3 shows testability results as proposed after the original application of STA, but before our testability enhancement and optimization are applied, whereas Table 4.4 shows results after testability enhancement and BIST optimization by test response redirection and test patterns sharing. In Table 4.3, the first column shows the names of the designs, the second column shows the number of TPGs, and the third column shows the number of MISRs**.** The fourth column depicts the percentage of operations that are fully controllable, the fifth column gives the percentage of operations that are observable and the sixth column shows the percentage of operations that are testable. The BIST registers reported in Table 4.3 are obtained by assuming that all primary input registers and primary output registers are converted to TPGs and MISRs respectively.

The total number of TPGs, MISRs and BILBOs after enhancement to 100% testability is shown in Table 4.4 whose first column depicts design names and number of test sessions needed to test them after 100% testability has been achieved by using our approach. The sub-columns of the column titled Straightforward show the number of TPGs, MISRs and BILBOs after initial straightforward testability enhancement is performed. By straightforward testability enhancement we simply mean applying the testability enhancement algorithms discussed in Section 4.2 to get 100% testability. The sub-columns of the column titled Optimized show the number of TPGs, MISRs and BILBOs that remain in the design after our BIST resource optimization by test response redirection and test patterns sharing are applied. The sub-columns of the column titled %Hardware reduction show

hardware reduction, in terms of BIST cost and number of BIST registers, resulting from our optimized approach as compared to the straightforward solution. In all our experimental results we have considered that T_{req}=T_{init}. The results show that by careful BIST optimization at the high-level, the needed BIST area overhead can be reduced by up to 15.4% and 14.3% in terms of hardware cost and the number of BIST registers respectively (last column in Table 4.4).

The second set of experimental results demonstrates our test-time constrained BIST resource optimization by test schedule shrinking and test schedule stretching. We will use the design Paulin (Table 4.5) to discuss our experimental results in detail and after that we will present a table, which summarizes the results for other designs. In Table 4.5 the column T_{req} is the requested testing time constraint and T_{BIST} is the testing time returned by our approach. Columns *TPG, MISR, BILBO, and CBILBO* represent the numbers of respective types of BIST registers. The column *Muxs* represents the number of test multiplexers and the column *Mux input* is the number of inputs of the test multiplexers. The column *HW cost* is the sum of the area of the BIST registers and test multiplexers.

Table 4.5 Optimization by test schedule shrinking and stretching for the design Paulin

T_{req}	T_{BIST}	TPG	MISR	BILBO	CBILBO	Muxs	Mux inputs	HW Cost	
1	1	2	3	0	7	6	12	492000	Shrink
2	2	4	3	2	2	2	4	354000	,,
3	3	3	3	2	2	4	8	338000	,,
4	4	4	3	0	3	2	4	324000	,,
5	5	4	3	2	1	2	4	304000	,,
6	6	5	3	2	0	0	0	270000	straightforward
6	6	4	3	2	0	2	4	254000	Optimized by sharing MISRs+TPGs
7	7	4	1	1	0	4	11	159500	Stretch
8	***7***	***4***	***1***	***1***	***0***	***4***	***11***	***159500***	***„***
9	***7***	***4***	***1***	***1***	***0***	***4***	***11***	***159500***	***„***
10	***7***	***4***	***1***	***1***	***0***	***4***	***11***	***159500***	***„***

Initially, after testability enhancements were applied, the resulting number of test sessions, T_{init}, for the Paulin example, was 6.

We have progressively shrinked the test schedule (T_{req}=*5, 4... 1*) and observed how BIST resources are utilized for each requested number of test sessions. It is observed that more BIST resources are needed to shrink the test schedule. The shorter the schedule the higher the BIST cost needed to guarantee it.

We have also performed experiments on resources optimization by test schedule stretching. It is observed that we can reduce the number of BIST registers by elongating the test schedule. However, additional multiplexers are needed to guarantee the testability (by BIST resource sharing). Stretching the test schedule beyond *7* test sessions does not result in any more reduction in BIST cost. This is due to the fact that any more reduction in BIST registers by relaxing testing time constraints renders the design untestable or test multiplexers add more overhead than can be gained by recovering BIST registers. Therefore, there is no benefit to stretch the test schedule beyond *7* test sessions. Our approach shows that if a minimal amount of BIST resources to make the design testable is used the design must be tested in 7 test sessions. In both cases, shrinking and stretching the test schedule, our approach tries to optimize hardware resource cost (BIST and multiplexer) in such a way that the design can be tested in a requested number of test sessions.

Figure 4.10 shows how BIST cost changes as the test schedule is shrinked or stretched to satisfy testing time constraints for the design Paulin. Figure 4.11 shows the relationship between the percentage changes in BIST cost, as the test schedule is shrinked or stretched for the same design. The reference BIST cost is the cost that our initial testability enhancement achieves after performing further hardware optimization by test patterns sharing and test responses redirection, but before test schedule stretching or shrinking are performed. It is observed that the BIST cost can increase by 94% as we try to optimize the BIST so as to test the design in one test session.

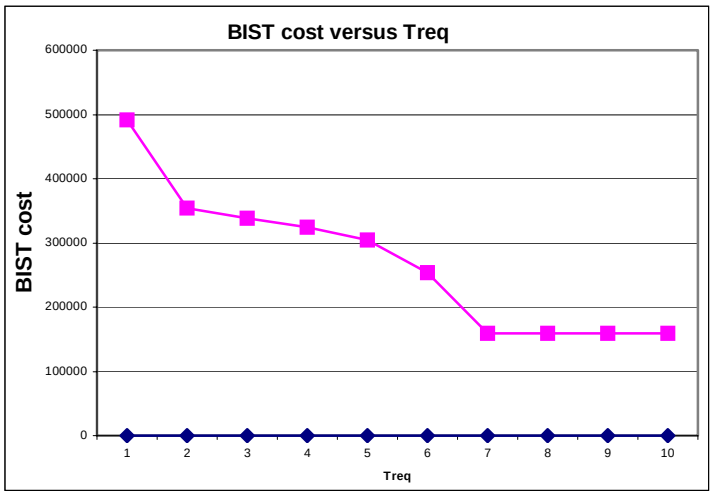


Figure 4.10 BIST cost versus testing time

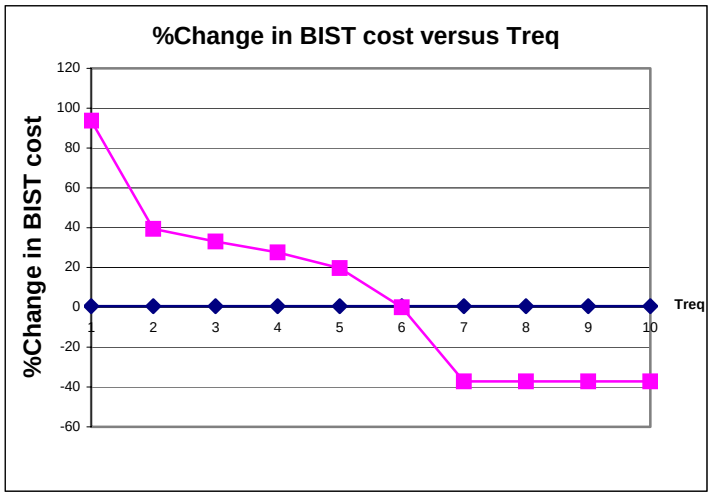


Figure 4.11 Percentage change in BIST cost versus testing time

We have tested several other designs and the results are depicted in Table 4.6. The first column gives design names. The second

column represents Δshr_{1TS} and the third column represents Δstr_{thr}. These terms are defined in the following two equations:

$$\Delta shr_{1TS} = \frac{shr_{1TS} - opt}{opt} \times 100\% \qquad (4.2)$$

$$\Delta str_{thr} = \frac{str_{thr} - opt}{opt} \times 100\% \qquad (4.3)$$

where shr_{1TS} is the hardware cost that is needed if the test schedule is shrinked so that the design is tested in 1 test session and str_{thr} is the smallest hardware cost which our stretch test schedule algorithm obtains assuming that it can stretch the test schedule infinitely long. On the other hand, *opt* is the optimized hardware cost obtained after applying our test pattern sharing and test response redirection algorithms, but not test schedule shrinking or stretching. In other words Δshr_{1TS} is the percentage change in the BIST cost if the test schedule is shrinked so that design is tested in 1 test session as compared to the optimized cost after test pattern sharing and test response redirection.

Table 4.6 Comparison of the results

Design Name	Δshr_{1TS} **(single TS)**	Δstr_{thr}	T_{init}	T_{thr}	T_{max}
Paulin	94	-37	6	7	10
Real	82	-35	3	5	11
Overnctrl	87	-21	5	6	8
Tseng	66	-35	4	5	8
EWF	173	-31	7	9	34
Average	**100,4**	**-31,8**			

The column named T_{init} is the initial number of test sessions as discussed in Sub-section 4.5. T_{max} is the required number of test sessions if one operation is to be tested per test session. We have observed that, on average, an increase in BIST cost of 100% is needed to be able to test the design in 1 test session. On the other hand, an average BIST cost reduction of 32% is achieved if we relax our testing time constraint to be very large. Experimental results show that in all designs, stretching the test schedule by

providing very large testing time constraints is not necessarily beneficial in terms of BIST hardware reduction. Each design has a limit, T_{thr}, on the number of test sessions beyond which, further stretching does not lead to any more hardware reduction. The BIST hardware that can satisfy testing the design in T_{thr} test sessions is also the minimum hardware that is needed to guarantee testability of the design. Removal of any more BIST registers will make the design untestable or more expensive due to too many multiplexers, which will be needed.

Chapter 5

Wiring-Aware BIST Synthesis

This chapter describes a hardware overhead minimization technique used during a BIST synthesis process. The technique works at the RT level and inserts a minimal amount of BIST resources into a digital system to make it fully testable. It takes into consideration the cost of the functional modules, multiplexers, BIST registers and wiring in order to obtain the minimal area designs. The problem of optimizing BIST insertion at the behavioral and RT levels while taking into account geometrical information of the design has been formulated in Section 1.2. Since the problem is NP hard, two optimization heuristics, a simulated annealing (SA) algorithm and a greedy heuristic, are used to solve the overhead minimization problem. Experimental results show that considering wiring area during BIST synthesis results in smaller final designs in terms of silicon area as compared to the cases when the wiring impact is ignored.

5.1 Design Transformations for BIST

To influence testability, our approach modifies the design by inserting BIST components. A number of BIST design transformations (testability modification moves) have been defined.

The transformations provide the optimization heuristic with a mechanism to perform neighborhood search so as to converge towards low area, self-testable designs. Some of the transformations can enhance testability while others can reduce

it. The testability reducing transformations enable the optimization strategy to create intermediate solutions, which later, as a result of further transformations, converge towards minimal area designs while guaranteeing testability. Therefore, for each type of testability enhancement transformation (TET) a reverse testability enhancement transformation (RTET) to cancel its effect is also defined. For example, if $TET(r_k,TPG_k)$ is a move that converts a functional register r_k, to a TPG_k, then its reverse transformation, converts TPG_k to functional register r_k, and is used to eliminate the effect of $TET(r_k,TPG_k)$.

Each testability transformation has advantages (reduced area overhead, reduced wiring, and/or improved testability) or disadvantages (additional area overhead, additional wiring, and/or reduced testability).

Some transformations introduce additional interconnections, which are used to connect the BIST modules to untestable data-path modules. These interconnections introduce area overhead. Furthermore, accommodating newly added BIST modules or test multiplexers can cause changes in positions of other modules on the chip. This can happen if the floor-plan algorithm changes the position of the modules to accommodate newly added ones in an efficient way. Such changes can impact wiring length and, hence, total design area.

5.1.1 Types of BIST Transformations

Our design transformations for BIST are classified in four types: conversion for controllability, conversion for observability, connection for controllability and connection for observability.

5.1.1.1 Conversion for Controllability

Transformations of this type change controllability by converting existing functional or signature analysis registers to BIST registers, which generate on-chip test patterns.

Converting an existing functional register to a TPG improves controllability of its output interconnections and can also improve controllability of other nodes further below it in the design. Since the TPG usually occupies larger area than a functional register, the conversion incurs an additional area overhead $(\delta A=A_{TPG}$-

$A_{register}$). The reverse transformation that converts the TPG back to the functional register is also defined. While the resulting functional register occupies smaller area, the controllability can be impaired when the reverse transformation is applied.

A MISR can be enhanced to a BILBO in order to give it both capabilities to generate test vectors and analyze test responses at different times. The disadvantage of the BILBO is its larger area compared to the MISR. A reverse transformation that converts the BILBO to the MISR is also defined. It reduces controllability.

If it is necessary to improve test application time, the BILBO can be enhanced to a CBILBO, which can simultaneously perform both test pattern generation and test response compression. The CBILBO is much larger than the BILBO. A reverse transformation from the CBILBO to the BILBO is also defined. The transformation affects testing concurrency, but not testability.

5.1.1.2 Conversion for Observability

Transformations of this type improve observability by converting some of the existing functional or test pattern generation registers to signature analysis registers. Three types of BIST registers can be used to enhance the observability. They are MISRs, BILBOs and CBILBOs.

To improve the observability, an existing functional register can be converted to a MISR, which can only compress test responses to a single test signature. A MISR occupies larger area than the functional register of equivalent bit-width. A move is also defined to convert a MISR to a functional register in order to recover area. Unfortunately, this reduces on-chip test response analysis capability. Similarly, the TPG can be enhanced by converting it to a BILBO or CBILBO, which performs the dual job of generating test patterns and compressing test responses on the chip. A reverse transformation that converts the BILBO or CBILBO to the TPG is defined. It can save area, but observability can be reduced.

5.1.1.3 Connection for Controllability

This is a controllability enhancement transformation whereby an existing TPG, BILBO or CBILBO is connected directly to an input of the functional module. Allocation information is used to analyze

the impact of the transformation on the corresponding SDFG. The impact of the connection made in the RTL design is translated to the corresponding connections in the SDFG by the help of the allocation/binding information. After the corresponding SDFG connections are identified, corresponding *variable-to-operation* connections in the SDFG are also made. To determine testability after the transformation, STA is performed on the transformed SDFG. The move necessitates addition or expansion of a *test multiplexer* in front of the RTL functional module in order to bring test patterns to the module in the test mode. Thus, wiring and multiplexer costs are increased ($\delta A = A_{wire} + A_{mux}$). The move can improve controllability of any RTL module since any module can be accessed by a direct connection.

The reverse transformation, which disconnects the TPG, the BILBO or the CBILBO from the functional module, results in reduced testability, but wiring can be removed and the number of inputs to a test multiplexer can be reduced. If, after the disconnection is performed, only one input remains, then the whole test multiplexer is removed.

5.1.1.4 Connection for Observability

This is an observability enhancement transformation that connects the output of a module directly to an existing MISR, BILBO or CBILBO. The transformation introduces wiring and expands a test multiplexer with an additional input or adds a new multiplexer. It can enhance the observability of any RTL module since any module can be connected to an existing MISR, BILBO or CBILBO.

The reverse transformation, which disconnects the MISR, BILBO or CBILBO from the module, results in reduced observability, but savings in wiring and at the same time reductions in the number of test multiplexer inputs can be achieved. As discussed earlier in this section, allocation information is used to translate the effect of the transformation on the corresponding SDFG on which testability analysis is performed.

5.1.2 Transformation Illustration and Motivational Examples

In this Section we will illustrate the transformations for BIST by using examples.

Suppose, for the SDFG in Figure 5.1a, one adder implements the operations $+_1$ and $+_2$ and one multiplier implements the operation *. One example of an RTL data-path implementation that satisfies this allocation constraint is shown in Figure 5.1b, whereby a register R_1 implements variables *a, e, g*; R_2 variables *b, f*; R_3 variable *c* and R_4 variable *d*.

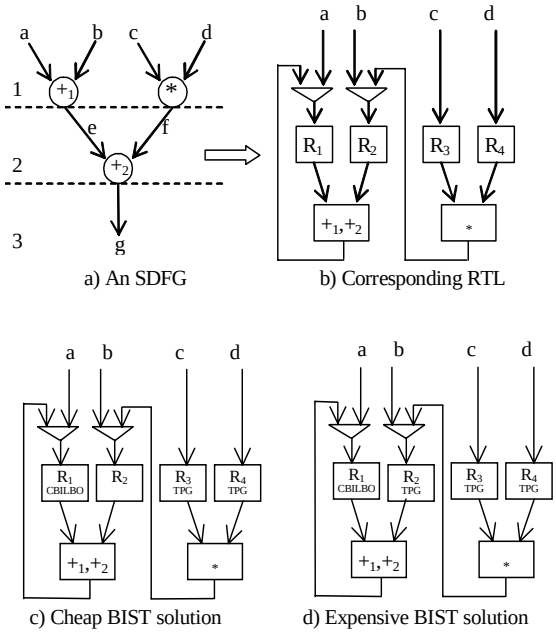


Figure 5.1 Illustrating conversion transformations

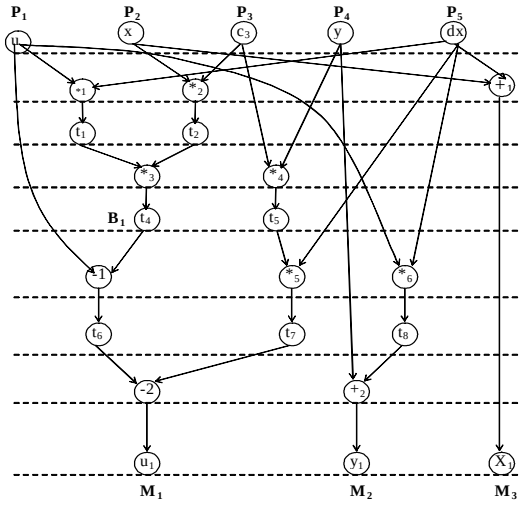
Operation $+_1$ can be tested by generating test patterns from variables *a* and *b*, constraining variable *c* to *0* and observing test responses on variable *g* (see Figure 5.1a). Similarly, the operation $+_2$ can be tested by supplying test patterns from variables *a* and *c*, constraining variable *b* to 0, and *d* to 1. Test responses are observed on variable *g*. The operation * is tested by supplying test patterns from variables *c* and *d*, constraining variables *a* and *b* to *0*, and observing test responses on variable *g*.

Since one adder implements the operations $+_1$ and $+_2$, it is sufficient to test only one of them to have the adder tested. To test

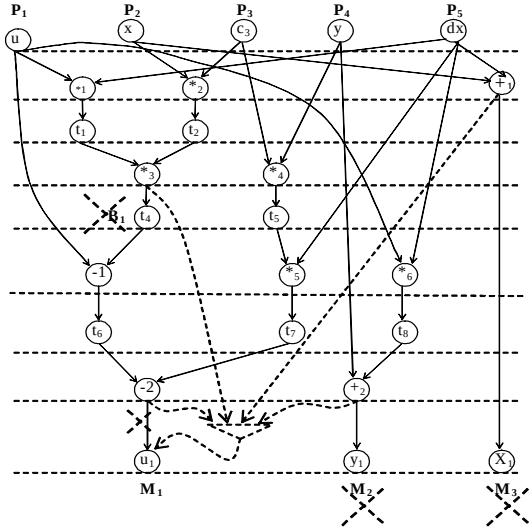
the multiplier, the operation * is tested. Suppose that to test the RTL design, we choose to test the operations $+_2$ and *. To test $+_2$ implies that registers R_1 *(variable a)* and R_3 *(variable c)* are converted to TPGs; R_1 *(variable g)* is converted to MISR; Since R_1 (*variable a*) is converted to TPG and R_1 (*variable g*) to MISR, R_1 has to be converted to CBILBO instead. In this way R_1 can generate test patterns and analyze test responses at the same time. To test the * operation, implies that R_3 *(variable c)* and R_4 *(variable d)* are to be converted to TPGs and R_1 *(variable g)* to MISR; since R_1 is to be converted to CBILBO for testing $+_2$, it is not converted to MISR for testing *. R_1 remains as CBILBO which can be used to test both $+_2$ and *. Thus, if R_1 is transformed to CBILBO, R_3 to TPG and R_4 to TPG, then the RTL design can be self-tested by using the original dataflow control flow, see Figure 5.1c. These register transformations for BIST are used in our optimization approach.

On the other hand, if the operation $+_1$ is used to test the adder, R_1 *(*variable *a)* and R_2 *(*variable *b)* are converted to TPGs and R_1 *(*variable *g)* to MISR. Since R_1 is already a CBILBO it is not converted to MISR. Thus, to test the RTL design 3 TPGs and 1 CBILBO are needed, see Figure 5.1d. This is more expensive than if operations $+_2$ and * are used. In this case our optimization heuristic chooses $+_2$ and * to test the design, which leads to a cheaper design with only 2 TPGs and 1 CBILBO, Figure 5.1c.

Now let us use the design in Figure 5.2a to illustrate the wiring connection transformations. Assume a one-to-one SDFG to RTL allocation so that the SDFG is the same as the RTL. After STA is performed, all primary inputs and constant nodes *(u, x, c_3, y, dx)* are converted to TPGs *(P_1, P_2, P_3, P_4, P_5)* and all primary outputs *(u_1, y_1, x_1)* to MISRs *(M_1, M_2, M_3)*. An internal register, t_4, is converted to a BILBO B_1 to enhance observability of the sub-graph consisting of operations **1, *3, *2* and to enhance controllability of the sub-graph consisting of operations *-1* and *-2*. Figure 5.2a shows one example of a self-testable version of the RTL design after testability modifications. More transformations can be performed to get an even cheaper design.



a) An SDFG after conversion transformations



b) An SDFG after successive connection transformations

Figure 5.2 Illustrating connection transformations

To find a cheaper solution using "*connection for observability*" transformations, one can connect the output of operation **3* through a multiplexer to the MISR M_1 and the BILBO B_1 is converted to a TPG. This intermediate design can be further transformed by connecting $+_1$ and, later, $+_2$ through the multiplexer to the MISR M_1. These successive transformations lead to the removal of more MISRs (M_2, M_3) since the operations $+_1$ and $+_2$ can now be observed using the MISR M_1. Since the multiplexer already exists, sharing of the MISR depends only on the trade-off between wiring and the MISR costs, which our optimization heuristic can find out. Figure 5.2b depicts the same design after successive transformations are applied. The dashed lines show the connection transformations. The crosses show the MISRs that were converted back to functional registers and the BILBO that was converted to the TPG.

In this example we have assumed that the wiring cost was cheaper compared to the MISR cost, which is why we eliminate the MISRs. However, a BIST synthesis optimization algorithm should use an appropriate cost function to decide which transformations to apply. In sections 5.4 and 5.5 we will describe two different approaches to optimize BIST resources.

5.2 Wiring Area Estimation Techniques

The wiring area is estimated using the heuristics presented in [5] and [29]. These heuristics first estimate the placement of modules on the chip based on the interconnection relationship and their sizes, and then compute the lengths of all interconnections. Finally, the wiring area, A_w, is computed using equation (5.1).

$$A_w = \left(\left(\sum_i l_i \times w_i \times W_{av} \right) - k_{or} \times A_{nodes} \right) / N_{metal} \qquad (5.1)$$

The sum over i is for all interconnections. l_i, and w_i are the length and width (in bits) of interconnection i, W_{av} is the average width of wires in the design (including the space needed between them), k_{or} is the over-route factor (fraction of the area above the datapath nodes and controller units in the metal layers that can be used for

routing), A_{nodes} is the area of the datapath nodes, and N_{metal} is the number of metal layers available.

The input to the wiring area estimation algorithm consists of a list containing the area of each data-path node to be placed, a list of interconnection relationships between the RTL nodes and constants W_{av}, k_{or} and N_{metal}.

5.3 Cost Function

The objective of the BIST synthesis process is to minimize the total area of the design, which is the sum of the areas of functional modules (A_{fmod}), functional registers (A_{freg}), functional multiplexers (A_{fMUX}), BIST registers (A_{BIST}), test multiplexers (A_{tMUX}) and wiring area (A_{wire}) as depicted in equation (5.2). A_{total} in the equation (5.2) is the cost that is used to drive our BIST optimization heuristics.

$$A_{total} = A_{freg} + A_{f\,\mathrm{mod}} + A_{fMUX} + A_{BIST} + A_{tMUX} + A_{wire} \qquad (5.2)$$

It is assumed that the number of functional modules is fixed when BIST insertion is performed. In other words, this means that the HLS algorithm performs scheduling and allocation before BIST insertion is performed.

5.4 BIST Synthesis Optimization with Simulated Annealing

In this Section, a simulated annealing algorithm [60] has been used to optimize the BIST structures in order to minimize the total design area. The overall optimization approach is assisted by a testability analysis, which identifies hard to test operations and modules. The testability analysis is performed at the behavioral level on the SDFG representation of the design. The BIST synthesis optimization approach can result in very good designs in terms of area since both geometrical information and testability are simultaneously taken into account during the synthesis process. On the other hand, since computation intensive

testability analysis and wiring area estimation are performed in each optimization iteration, the whole approach can be slow.

Testability enhancement is performed on the RT-level representation of the design. The simulated annealing optimization process uses testability transformations to explore a design space in search of the smallest self-testable design (see Section 5.1). Since our objective is to minimize the total design area, high degree of BIST register sharing is also achieved.

After testability enhancement is performed, 100% controllability, observability, and testability are achieved for all the modules. Therefore, for each module, 100% fault coverage is also achieved, provided that the modules do not have random resistant faults and a sufficiently large number of pseudo-random test patterns is applied. If modules have random resistant faults, they need to be made random resistant free by test point insertion. A low overhead technique for removing random resistance is described in [22]. If this technique cannot be deployed, to cover random resistant faults, a few deterministic test vectors must be used at the end of the pseudo-random test session. This work assumes that all modules are random resistant free, hence 100% fault coverage can be achieved without test point insertion or deterministic test vectors.

5.4.1 Simulated Annealing Based BIST Synthesis Framework

A simulated annealing based BIST synthesis optimization framework is depicted in Figure 5.3. In our implementation of the simulated annealing, a fully testable initial solution, X_0, is selected as an input to the heuristic. A fully testable initial solution is generated by conversion transformations. All constant nodes and PI registers are converted to TPGs, all PO registers to MISRs and all internal registers to BILBOs. This gives a very expensive fully testable initial solution.

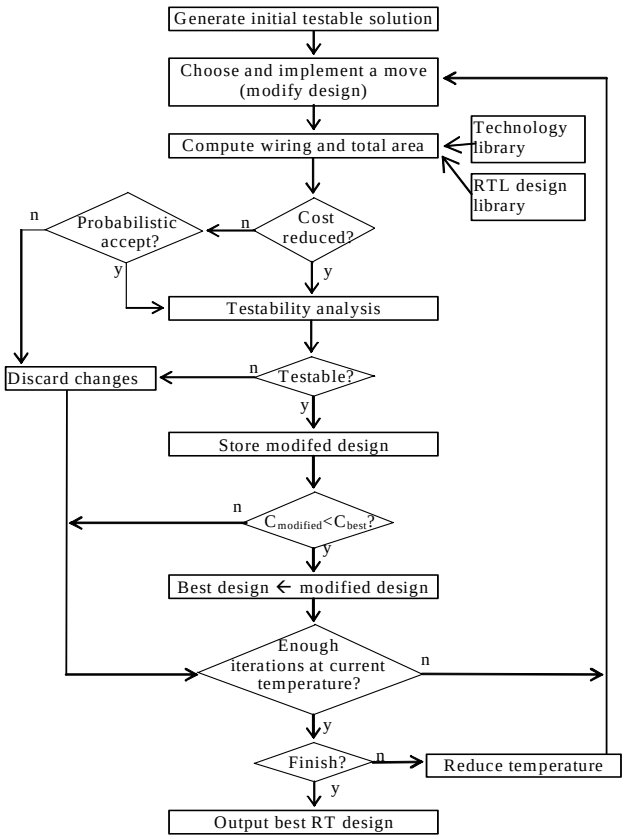


Figure 5.3 Simulated annealing BIST synthesis framework

Neighboring solutions are generated by randomly applying testability transformations on the design. The cost of the design is computed as discussed in Section 5.3. The selected transformation is used to modify the design to produce a neighboring design, which is referred to as the *neighboring solution*. Since our aim is to converge towards minimal area, it is obvious that expensive transformations that increase area cost should be rejected. However, in order to escape from a local optimum, the simulated annealing approach [60] can probabilistically accept expensive transformations hoping that

subsequent transformations may enable the optimization strategy to converge towards the global optimum of the cost function. As the temperature parameter decreases, probabilities of accepting expensive transformations decrease as well [60], hence at low temperatures only the cheap solutions are accepted and the annealing converges to minimum area designs. After applying the transformation, if the area of the design is reduced or the transformation can be probabilistically accepted, then the testability status (testable, not testable) of the RTL design is checked by performing STA on the corresponding SDFG representation. The optimization algorithm proceeds as depicted in Figure 5.3.

5.4.2 Experimental Results

Our approach does not only show how many TPGs, MISRs, BILBOs and CBILBOs are added, but also performs quantitative estimation of the wiring cost during the BIST synthesis process. It takes overall design cost as the optimization objective. Thus, it potentially results in more cost efficient designs. Other approaches use the number of TPGs, MISRs, BILBOs and CBILBOs as optimization criteria. Since they ignore quantitative computation of wiring area, they do not necessarily guarantee highly efficient designs in terms of total design area.

We have tested our approach on several benchmarks. In our experiments, the technology dependent parameters are based on Intel's 65nm logic technology [7], which will be delivered for production in 2005. In this technology a transistor occupies an estimated area of $0.1\mu m^2$. We have assumed that the RTL modules will be wired using metal layer 3, 4 or 5, and that we will have two of these layers to use for connecting the RTL modules. The wiring pitch (the average width of a 1-bit wire including spacing between the wires) used is $0.277\mu m$. It is the average pitch of the metal layers 3, 4 and 5, whose respective pitches are $0.22\mu m$, $0.28\mu m$ and $0.33\mu m$. We have assumed that wire over-routing factor is 0.5.

Characteristics of the designs have already been presented in Section 4.7. To estimate sizes of the functional registers, functional modules, multiplexers and BIST registers, we have developed a library of gate-level RTL modules. To compute the

number of transistors in basic gates such as AND, NAND, OR, NOR, XOR and NOT, we have used an approach described in [35]. The number of transistors that are required to implement a given RTL module is obtained by summing up the number of transistors in all the gates which compose the module. The size of the modules is then computed as the product of the size of a transistor in the 65nm technology and the total number of transistors in the module. The number of transistors required to implement 16-bit modules are: adder – 480, subtractor – 704, multiplier – 8736, divider – 9248, register – 96, TPG – 210, MISR – 306, BILBO – 376 and multiplexer – $96\times(N-1)$, where N is the number of multiplexer inputs.

The experimental results are summarized in Table 5.1 through Table 5.3. Columns *P, M* and *B* denote the number of TPGs, MISRs and BILBOs respectively. A_{nodes} is the sum of the node areas, A_{wires} is the sum of all wiring areas, N_{mux} is the number of test multiplexers, A_{mux} is the area of test multiplexers and $\%A_w/A_t$ is the percentage of wiring area with respect to total design area. A_{total} represents the total area of the design after synthesis, including wiring area. The column with title *time* represents CPU time taken by our approach. Experiments were run on a Sun Solaris workstation with 440 MHz CPU and 256MB RAM.

In Table 5.3, $A_{t\_with\_wire}$ and $A_{t\_no\_wire}$ are the respective total design areas with and without considering wiring during BIST optimization. The third column represents the percentage of unnecessary area overhead (UAO) which is incurred if wiring is not considered as compared to the case when wiring is considered.

The optimization process was run in two different ways. In the first case we have ignored wiring area ($A_w=0$) during our BIST synthesis optimization. This is in accordance with many previous works that optimize BIST by only counting the number of TPGS, MISRs and BILBOs that are introduced. The design cost that is minimized is the total area consisting of functional data-path nodes, BIST registers and test multiplexers. The results are shown in Table 5.1. In the second set of experiments, we have taken into account wiring area during the BIST optimization process, as described in this section. The cost minimized is the total area of the data-path nodes, BIST nodes, test multiplexers and *wiring area*. The results

are summarized in Table 5.2. For each case we have run our total area estimation algorithm to compute total area (wiring inclusive) after synthesis.

Table 5.1 Wiring area ignored during optimization

Design Name	P	M	B	A_{nodes} (μm^2)	A_{wire} (μm^2)	A_{total} (μm^2)	$\%A_w/A_t$	N_{mux}	A_{mux}	CPU Time (sec)
Paulin	4	1	1	5841.8	3072.1	8913.9	34.5	8	124.8	3412.7
Real	3	1	1	6019.2	3508.9	9528.1	36.8	12	144.0	4609.0
Overnctrl	4	1	0	2444.2	2537.2	4981.4	50.9	10	134.4	3259.2
EX2	3	1	0	4794.4	2362.8	7157.2	33.0	9	105.6	2162.6
EWF	4	1	6	9076.2	8773.1	17849.3	49.2	15	192.0	10348.3

Table 5.2 Wiring area considered during optimization

Design Name	P	M	B	A_{nodes} (μm^2)	A_{wire} (μm^2)	A_{total} (μm^2)	$\%A_w/A_t$	N_{mux}	A_{mux}	CPU Time (sec)
Paulin	5	3	1	5770.4	1284.2	7054.6	18.2	0	0	1439.1
Real	6	2	4	6014.4	1391.6	7406.0	18.8	0	0	2017.3
Overnctrl	6	2	1	2381.6	1019.9	3401.5	30.0	0	0	1549.5
EX2	6	1	1	4751.0	686.2	5437.2	12.6	0	0	1823.4
EWF	9	7	7	9095.2	5380.9	14476.1	37.2	0	0	5819.9

Table 5.3 Unnecessary area overhead if wiring is not considered

Design Name	$A_{t\_no\_wire}$ (μm^2)	$A_{t\_with\_wire}$ (μm^2)	UAO
Paulin	8913.9	7054.6	26.4
Real	9528.1	7406.0	28.7
Overnctrl	4981.4	3401.5	46.5
EX2	7157.2	5437.2	31.6
EWF	17849.3	14476.1	23.3

In our simulated annealing experiments, the temperature was decreased very slowly according to the formula $f(t) = t/(1+t*tscale)$ [60], where the parameter *tscale* is a suitably small value. We have set the parameter *tscale* to be 0.1 and the initial temperature to be 227. In our experiments we have not waited for the temperature to

become zero before stopping, instead we have set a stopping criterion that puts an upper limit on the number of consecutive rejected moves and the total number of simulated annealing iterations. The number of consecutive rejected moves is the number of consecutive simulated annealing iterations that are run, but which give no improvement in quality of the solution. We defined a constant known as the maximum number of allowed consecutive rejected moves (MNACRM). If the number of consecutive rejected moves exceeds the value of MNACRM, then our simulated annealing process terminates. We have set the value of MNACRM to be 1000.

For all designs, when the area of the wiring is taken into consideration during BIST synthesis optimization, we get smaller total design area, as shown in Table 5.3. If wiring is taken into consideration, the area occupied by the BIST registers is larger than that occupied by the BIST registers if wiring is not considered.

We compared the total node (functional, BIST and multiplexer) areas in the cases when wiring is considered and when it is not considered. The results show that when wiring area is considered, more BIST registers are used (Table 5.1, Table 5.2). This means that one would expect the total data-path node (functional + BIST) area to be larger in the case when wiring is considered and the gain in saving total area comes only from savings in wiring area, i.e. a trade off between BIST register and wiring is made. This is not the case in the presented experimental results, except for the design EWF. This is because when wiring is not considered some multiplexers are added. Our approach optimizes total design area (wiring, functional, BIST and multiplexers) in such a way that a globally cheaper design is generated.

5.5 BIST Synthesis Optimization with Greedy Heuristic

In this Section, we propose a greedy heuristic for addressing the problem of wiring-aware BIST synthesis optimization. The technique uses our behavioral and RT levels BIST enhancement

metrics to guide BIST synthesis. As discussed in the previous section, testability analysis is performed on the SDFG and testability enhancement based on BIST design transformations (see Section 5.1) is performed on the corresponding RTL architectural implementation.

The testability enhancement performed by our heuristic guarantees complete testability of each RTL module while keeping the design area minimum. The heuristic also addresses the drawbacks of the simulated annealing based BIST synthesis approach (discussed in Section 5.4), which is very slow.

The heuristic provides a novel way to quickly explore the design space in search of cheap, yet testable design solutions. It proceeds in the following steps:

A. Controllability enhancement.

B. Observability enhancement.

C. Global testability enhancement.

In each of these steps, the following two actions are repeated until complete controllability (step A), observability (step B) and testability (step C) are, respectively, achieved:

i. Choose a module *m* that is not controllable (observable, testable respectively).

ii. Visit all possible enhancements for the module *m* and choose the enhancement that incurs the lowest area overhead.

In order to make the design space exploration efficient, it is important to choose and enhance the modules in such a sequence so as to minimize the overall number of testability enhancements. This is made possible by using our novel *BIST enhancement metrics* (Section 5.5.1) to help decide in which sequence to enhance modules, which have controllability, observability, or testability problems.

5.5.1 BIST Enhancement Metrics

We need to choose uncontrollable, unobservable or untestable modules and an order in which to enhance them, in such a way that the total number of enhancements performed on the design is

reduced. We can achieve this objective by ensuring that each time we choose a module to enhance, the enhancement will improve as many other modules as possible.

To solve this problem, we propose an approach, which uses our novel behavioral-level BIST enhancement metrics to guide the testability enhancement process. The *BIST enhancement metrics* are defined below.

Definition 5.1: *Total Controllability Enhancement Potential (TCEP)* of a given SDFG operation or variable node is the number of operations and variables whose controllability it can affect.

Controllability of a node n_j can be affected by the controllability of a node n_i if there is a path in the SDFG from n_i to n_j and the control step of n_i precedes the control step of n_j. For instance, consider the operation **1* in Figure 5.4. It can be observed that starting from the operation **1*, it is possible to reach seven nodes namely t_1, **3*, t_4, *-1*, t_6, *-2*, and u_1. Therefore, the value of *TCEP* for the operation **1* is 7.

Definition 5.2: *Total Observability Enhancement Potential (TOEP)* of a given SDFG operation or variable node is the number of operations whose observability it can affect.

Observability of a node n_i can be affected by the observability of a node n_j if there is a path in the SDFG from n_i to n_j and the control step of n_i precedes the control step of n_j. For instance, consider the operation *-1* in Figure 5.4. It can be observed that starting from the operation *-1*, it is possible to traverse the graph upwards and reach 10 nodes namely *u*, t_4, **3*, t_1, **1*, *dx*, t_2, **2*, *x*, and c_3. Therefore, the value of *TOEP* for the operation *-1* is 10.

The BIST enhancement metrics *TCEP* and *TOEP* presented so far are computed with reference to the SDFG nodes. Controllability, observability and testability enhancements are, however, performed on the RT level architectural representation of the design. Therefore, we need to extend the definitions of the BIST enhancement metrics so that we can apply them to the RTL designs.

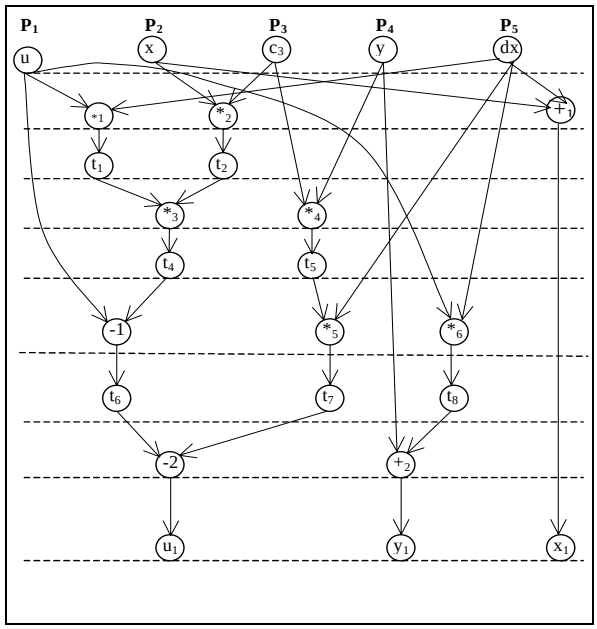


Figure 5.4 An SDFG to illustrate the BIST enhancement metrics

Definition 5.3: *RTL Total Controllability Enhancement Potential (RTCEP)* of a given RTL module, m_i, which implements a set of SDFG operations $SM_i = \{op_1, op_2 \ldots op_n\}$ whose respective values of the *TCEP* are given by the set $STCEP_i = \{TCEP_1, TCEP_2, TCEP_n\}$ is defined as the maximum *TCEP* value in the set $STCEP_i$, i.e. $RTCEP_i = Max_{j=1}^{n}\{TCEP_j\}$.

Definition 5.4: *RTL Total Observability Enhancement Potential (RTOEP)* of a given RTL module, m_i, which implements a set of SDFG operations $SM_i = \{op_1, op_2 \ldots op_n\}$ whose respective values of the *TOEP* are given by the set $STOEP_i = \{TOEP_1, TOEP_2 \ldots TOEP_n\}$ is defined as the maximum *TOEP* value in the set $STOEP_i$, i.e. $RTOEP_i = Max_{j=1}^{n}\{TOEP_j\}$.

To explain our RTL BIST enhancement metrics, consider the example of an SDFG shown in Figure 5.4. If a *1-to-1* SDFG to RTL allocation is used, the *TCEP* and *RTCEP* metrics are the same.

Similarly, the *TOEP* and *RTOEP* metrics are the same (see Table 5.4).

Suppose that a more realistic allocation, as shown in row 1 and row 2 in Table 5.5, is used. Row 3 shows the *TCEP* values and row 4 shows the *TOEP* values for the SDFG. After applying the definitions above, the values of *RTCEP* and *RTOEP* are shown in rows 5 and 6 respectively in Table 5.5.

Table 5.4 BIST enhancement metrics: 1-to-1 mapping

Modules	**M1**	**M4**	**M5**	**M2**	**M3**	**M6**	**A1**	**A2**	**S1**	**S2**
Operation binding	*1	*4	*5	*2	*3	*6	+1	+2	-1	-2
TCEP	7	5	3	7	5	3	1	1	3	1
TOEP	2	2	5	2	8	2	2	5	10	17
RTCEP	7	5	3	7	5	3	1	1	3	1
RTOEP	2	2	5	2	8	2	2	5	10	17

Table 5.5 BIST enhancement metrics: realistic mapping

Modules	**Mult1**			**Mult2**			**Add1**		**Sub1**	
Operation binding	*1	*4	*5	*2	*3	*6	+1	+2	-1	-2
TCEP	7	5	3	7	5	3	1	1	3	1
TOEP	2	2	5	2	8	2	2	5	10	17
RTCEP	7			7			1		3	
RTOEP	5			8			5		17	

Once testability analysis has identified a set of modules that have to be enhanced, we use the BIST enhancement metrics in order to decide which particular module out of them is to be enhanced first. The actual metric we use is *RTCEP* for the case of controllability and *RTOEP* for the case of observability. For the case of controllability enhancement, our criterion is to prioritize enhancement of the module that has the greatest value of the *RTCEP* among all uncontrollable modules. Similarly, for the case of observability enhancement, we prioritize enhancement of the module that has the greatest value of the *RTOEP* among all unobservable modules. The exact testability enhancement is then performed by applying *BIST design transformations* on the RTL design as described in Section 5.1.

Let us now consider a more exact description of how our BIST enhancement metrics are used. We discuss the enhancement procedure with respect to controllability enhancement. Suppose that $SM_i = \{op_1, op_2, \ldots, op_n\}$ is a set of operations that are implemented by the RTL module M_i. Suppose also that the respective values of the *TCEP* for the operations in the set SM_i are given by the set $STCEP_i = \{TCEP_1, TCEP_2, \ldots, TCEP_n\}$. Since any RTL functional module M_i implements one or more SDFG operations, it follows that $|SM_i| \geq 1$ and $|STCEP_i| \geq 1$. Suppose that after testability analysis is performed on the design, the set of uncontrollable RTL modules is found to be $URT = \{M_1, M_2 \ldots M_m\}$. An uncontrollable RTL module $M_x \in URT$ is chosen to be enhanced if there is an operation $op_y \in SM_x$, which it implements such that the operation op_y has the greatest value of *TCEP* among all the operations that are in the union set $STCEP_1 \cup STCEP_2 \cup \ldots \cup STCEP_m$.

5.5.2 BIST Synthesis Heuristic

A general overview of our BIST synthesis heuristic is depicted in Figure 5.5. In the first step, all modules are made controllable, in the second step, all modules are made observable. After controllability and observability are enhanced, it is still possible that some untestable modules will remain (see discussion in Section 4.2). Therefore, in the third step, all modules are made testable.

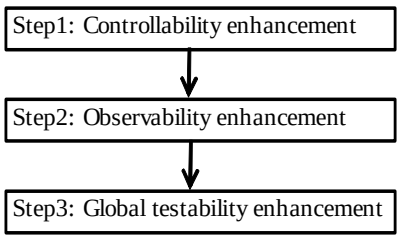


Figure 5.5 Steps of the BIST synthesis heuristic

The controllability enhancement algorithm, shown in Figure 5.6, takes as inputs a design represented in SDFG, allocation information and RTL data path, and returns a fully controllable RTL design. In a similar way, the algorithm depicted in Figure 5.7 is used to enhance observability.

Algorithm: EnhanceControllability

```
Begin
 1. Controllable ← False;
 2. while Controllable = False do
 3.     DFGEP ← ϕ;   RTEP ← ϕ;
 4.     UCP ← STA(G);
 5.     UCM ← UncontrollableModules(UCP, R, A);
 6.     if UCM = ϕ then
 7.         Controllable ← True;
 8.     else
 9.         for i ← 1, 2, .., |UCP| do
10.             ti ← GetTCEP(G, pi) | pi ∈ UCP;
11.             DFGEP ← DFGEP ∪ { ti };
12.         end for
13.         for i ← 1, 2, .., |UCM| do
14.             ti ← GetRTCEP(mi,DFGEP, A) | mi ∈ UCM;
15.             RTEP ← RTEP ∪ { ti };
16.         end for
17.         MTE ← ModuleToEnhance(UCM, RTEP);
18.         ψ ← ControllEnhancements (MTE, R);
19.         C ← ϕ;
20.         for i ← 1, 2,.., |ψ| do
21.             Ci ← EnhancementCost(Ei) | Ei ∈ ψ;
22.             C ← C ∪ { Ci };
23.         end for
24.         SE ← Ei ∈ ψ | cost(Ei) = Min_{j=1}^{|C|}{Cj} ;
25.         R ← Modify(R, SE);
26.     end if
27. end while
End.
```

Figure 5.6 Controllability enhancement

The symbols and notations used in the *pseudo-code* in Figure 5.6 are described as follows: *R* is the RTL data path, *G* is the corresponding SDFG of the design, and *A* is the allocation

information depicting the relationship between *G* and *R*. *UCP* and *UOP* are respective sets of all uncontrollable and unobservable operations. They are obtained by performing testability analysis of the SDFG. *UCM* and *UOM* are respective sets of all uncontrollable and unobservable RTL modules. They are computed based on the definition of RTL module controllability and observability.

Algorithm: EnhanceObservability

```
Begin
1.  Observable ← False;
2.  while Observable= False do
3.      DFGEP ← ϕ;  RTEP ← ϕ; UOP ← STA(G);
4.      UOM ← UnobservableModules(UOP, R, A);
5.      if UOM = ϕ then
6.          Observable ← True;
7.      else
8.          for i ← 1, 2,..,|UOP| do
9.              ti ← GetTOEP(G, pi) | pi ∈ UOP;
10.             DFGEP ← DFGEP ∪ { ti };
11.         end for
12.         for i ← 1, 2,  ..,|UOM| do
13.             ti ← GetRTOEP(mi, DFGEP, A) | mi ∈ UOM;
14.             RTEP ← RTEP ∪ { ti };
15.         end for
16.         MTE ← ModuleToEnhance (UOM, RTEP);
17.         ψ ← ObserveEnhancements(MTE, R);
18.         C ← ϕ;
19.         for i ← 1, 2, ..,|ψ| do
20.             Ci ← EnhancementCost(Ei) | Ei ∈ ψ;
21.             C ← C ∪ { Ci };
22.         end for
23.         SE ← Ei ∈ ψ | cost(E_i) = Min_{j=1}^{|C|}{C_j} ;
24.         R ← Modify(R, SE);
25.     end if
26. end while
Begin.
```

Figure 5.7 Observability enhancement

Procedure *GetTCEP(G, p_i)*, where $p_i \in UCP$**,** computes the *TCEP* value for the operation p_i. *DFGEP* is the set consisting of *TCEP* values of all the uncontrollable or unobservable operations in the SDFG. Procedure *GetRTCEP(m_i, DFGEP, A)*, where $m_i \in UCM$,

computes the *RTCEP* value for the module m_i. *RTEP* is the set consisting of *RTCEP* values of all the uncontrollable modules.

The procedure *ModuleToEnhance(UCM, RTEP)* searches for a suitable module to be enhanced, *MTE.* Procedure *ControllEnhancements(MTE, R)* returns ψ, which is the set of all possible enhancements for the uncontrollable module to be enhanced (*MTE*). The procedure *EnhancementCost*(E_i) returns the cost of applying the enhancement E_i. *C* is a set, which stores the costs of all the potential enhancements for the module *MTE.* The procedure *Modify(R, SE)* uses the selected enhancement $SE \in \psi$, to modify the RTL design.
Many of the notations used in the controllability enhancement algorithm are also used in the observability enhancement algorithm in Figure 5.7. In addition, the latter algorithm deploys a procedure *GetTOEP*(G, p_i), where $p_i \in UOP$, to compute the *TOEP* value for the operation p_i and procedure *GetRTOEP*(m_i, *RTEP, A*), where $m_i \in UCM$, to compute the *RTOEP* value for the module m_i. In the observability enhancement algorithm, the set *RTEP* consists of the *RTOEP* values for all the unobservable modules. *ObserveEnhancements(MTE, R)* is the procedure which finds all potential observability enhancements *(ψ)* for the unobservable module to be enhanced, *MTE.*

5.5.2.1 Enhancement Selection

We need to get the cheapest solution when a given module to enhance has been decided.

Let $M = \{m_1, m_2 \ldots m_k\}$ be a set of k functional modules that compose an RTL design. Suppose that *PTD* represents a *partially testable RTL design* at a certain moment during our controllability (observability or testability) enhancement process. Suppose that after testability analysis is performed a module $m \in M$ is selected for enhancement (see enhancement algorithm). Such a module can have multiple controllability (observability/testability) enhancement options that can be used. For example, convert its input register to a TPG or connect its input to an existing TPG or BILBO. Suppose that $E = \{e_1, e_2 \ldots e_n\}$ is a set consisting of n enhancements available for the module m. Each of the enhancements $e_i \in E$ is separately applied to the partially testable

design *PTD* to get a corresponding enhanced design d_i. Suppose that after these enhancements are respectively applied to the partially testable design *PTD,* the respective corresponding resulting enhanced designs form a set $D=\{d_1, d_2,\ldots, d_n\}$.

In order to decide which enhancement option *(BIST design transformation)* to use for the module *m*, we evaluate the cost of each improved partially testable design $d_i \in D$. Out of all the enhancements in the set *E*, the enhancement $e_i \in E$ that leads to the cheapest improved design is chosen. The cost that we use is the total design area, which consists of the areas of the functional modules, functional registers, BIST modules, test multiplexers as well as area contribution due to wiring.

5.5.2.2 Global Testability Enhancement and BIST Redundancy Minimization

After controllability and observability of all the modules are enhanced, it is still possible for some of them to be untestable. The first part, lines 1-27 of the algorithm shown in Figure 5.8, proposes a technique to fix the remaining testability problems.

In Figure 5.8, the symbol Ω represents the set of all enhancements that are done on the design. Procedure *UntestableModules(G, R, A)* takes the SDFG, the RTL design and allocation information, then uses STA to find a list of all the untestable modules, *UTM*. The first untestable module from the list *UTM,* denoted as *M*, is usually the first one to be enhanced.

Procedure *Enhance(M, operand, enhanceType)* adds a BIST enhancement for the module *M*. It is used to enhance output observability or controllability of the left or right input of the module. *DiscardEnhancement(R, Enh)* is used to remove the enhancement, *Enh,* from the design. Procedure *PutBackEnhancement(R, E_i)* puts back the enhancement E_i if its removal renders the design untestable.

Algorithm: EnhanceTestabilityAndMinimizeRedundantBIST

```
Begin
1.  Ω ←Set of all enhancements from Step1 and Step2 in Figure 5.5;
2.  UTM ← UntestableModules(G, R, A);
3.  while UTM != ϕ do
4.      M ← FirstUntestable(UTM);
5.      Enh ← Enhance(M, Left, Contr);
6.      UTM ← UntestableModules(G, R, A);
7.      if UTM != ϕ then
8.          DiscardEnhancement(R, Enh);
9.          Enh ← Enhance(M, Right, Contr);
10.         UTM ← UntestableModules(G, R, A);
11.         if UTM != ϕ then
12.             DiscardEnhancement(R, Enh);
13.             Enh ← Enhance(M, Output, Observ);
14.             UTM ← UntestableModules(G, R, A);
15.             if UTM != ϕ then
16.                 Enh ← Enhance(M, Left, Contr);
17.                 Enh1 ← Enhance(M, Right, Contr);
18.                 Ω ← Ω ∪ {Enh} ∪ {Enh1};
19.             end if
20.         else
21.             Ω ← Ω ∪ {Enh};
22.         end if
23.     else
24.         Ω ← Ω ∪ {Enh};
25.     end if
26.     UTM ← UntestableModules(G, R, A);
27. end while
28. // Remove unnecessary BIST overhead
29. for i ← 1, 2,..,|Ω | do
30.     Ei ← GetEnhancement | Ei ∈ Ω;
31.     DiscardEnhancement(R, Ei);
32.     UTM ← UntestableModules(G, R, A);
33.     If UTM != ϕ then
34.         PutBackEnhancement(R, Ei);
35.     end if
36. end for
End.
```

Figure 5.8 Global testability enhancement and redundant BIST minimization

After all the modules are enhanced and the design becomes testable, it is likely that we have added too much BIST overhead.

Therefore, we propose a BIST resources minimization (BIST redundancy removal) phase, whereby we try to remove each enhancement we have added and check if the design remains testable. If the design remains testable after the removal, the change is made permanent. Otherwise the enhancement is put back. In this way BIST overhead is reduced while testability is still guaranteed. Pseudo-code of our redundant BIST hardware removal algorithm is given as part of Figure 5.8 (lines 28-36).

5.5.3 Experimental Results

We have evaluated our heuristic on several benchmarks. The technology dependent parameters as well as the sizes of the RTL modules used are the same as those in the experimental results presented in Section 5.4. Characteristics of the designs we used in our experiments have already been summarized in Table 4.2.

Our experimental results are summarized in Table 5.6. Columns titled *P*, *M* and *B* represent the number of TPGs, MISRs and BILBOs respectively. The column titled *Design Area* represents the area of the designs before and after our BIST synthesis heuristic is applied. The column titled *overhead* shows the hardware overhead of our approach. The last column represents the CPU time taken by our heuristic. The experiments were run on a Sun Solaris workstation with 440MHz CPU and 256MB RAM.

Table 5.6 Experimental results using our heuristic

Design Name	**P**	**M**	**B**	**Design Area (μm^2)**		**Overhead (%)**	**CPU time (Sec)**
				Before	**After**		
Paulin	5	3	1	6915.1	7054.6	2.0	113
Real	6	2	4	7195.7	7406.0	2.9	199
Ovenctrl	6	2	1	3262.1	3401.5	4.3	70
Ex2	6	1	1	5329.8	5437.2	2.0	91
Ewf	9	7	7	14004.1	14476.1	3.4	1030
Average						2.92	

In our experiments, we have taken into account wiring area during the BIST optimization process, as described in this section. The

design cost minimized is the total data path area including the area of functional and BIST modules, test multiplexers and *wiring*.

The importance of considering wiring during the BIST synthesis process is already experimentally justified in Section 5.4. The importance of the work presented in this section is on getting a faster approach that can be applicable to realistic large designs instead of the slow simulated annealing based approach presented in Section 5.4. We have, therefore, compared the results of our greedy approach with the results of our simulated annealing based approach presented in Section 5.4. As it can be observed from Table 5.7, the proposed approach is efficient in terms of run time and, at the same time it also produces good quality results. While run times are on average one order of magnitude lower, the quality of the results produced by the heuristic is on average the same as that generated with the simulated annealing approach.

Table 5.7 Performance comparisons

	Simulated annealing (Wire considered)		**Our heuristic (Wire considered)**		
Design Name	**Area (μm^2)**	**CPU Time (Sec.)**	**Area (μm^2)**	**CPU Time (Sec.)**	**CPU Time reduction (#Times)**
Paulin	7054.6	1439.1	7054.6	113	12.7
Real	7406.0	2017.3	7406.0	199	10.1
Ovenctrl	3401.5	1549.5	3401.5	70	22.1
Ex2	5437.2	1823.4	5437.2	91	20.0
Ewf	14476.1	5819.9	14476.1	1030	5.7
Average					14.1

We have also tested our approach with a large design (LD), which was randomly generated. This design has 235 modules of which 83 are adders, 32 multipliers and 120 registers. Our greedy heuristic converted 5 registers to TPGs, 14 to MISRs and 24 to BILBOs to make the design testable. Overall hardware overhead due to our approach is only 2.05% and run time is 12783 seconds. When we tried to run simulated annealing with the large design, run times were so high that we terminated the program prematurely.

To further analyze the performance of the heuristic, we have split CPU times in two parts: the time taken by the testability analysis (STA time) algorithm and the time taken by the cost computation algorithm as depicted in Table 5.8. For the designs we have experimented with, STA time is far larger than the cost computation time. On the other hand, as the designs become very large, cost computation time can grow quickly and become comparable or even larger than the STA time. Since our heuristic restricts the number of STA invocations, the testability analysis time will not grow very fast. However, as the design becomes large, there is increasing number of possible testability enhancements for a given chosen functional module. Since each possible enhancement involves evaluation of the design area, these possibilities will put a heavy computation burden on the heuristic. Therefore, for large designs, a fast and accurate design area *estimation* algorithm is needed to overcome this problem.

Table 5.8 CPU time comparisons

Design Name	#STA runs	CPU time (Sec.)		
		STA	Design cost	Total
Paulin	43	109	4	113
Real	61	193	6	199
Overnctrl	24	67	3	70
Ex2	41	86	5	91
Ewf	112	898	132	1030
LD	448	11495	1288	12783

Chapter 6

Conclusions and Future Work

This thesis has proposed approaches to solve BIST synthesis problems that work at a high-level of abstraction. In this chapter a concluding summary and directions for possible future extensions of the work presented in this thesis are provided. The conclusions are presented in Section 6.1 and future research extensions in Section 6.2.

6.1 Conclusions

Improving testability of the designs by inserting BIST components adds hardware overhead. In this thesis we have provided approaches to minimize the amount of BIST hardware overhead while guaranteeing 100% testability. Symbolic testability analysis has been used to reveal hard to test parts of the design whose testabilities need to be enhanced.

In Chapter 4, a testing-time constrained approach to minimizing BIST overhead has been proposed. It explores alternative testability options that exist in the design to minimize the BIST hardware overhead while satisfying testing-time constraints and 100% testability.

Initially, the approach adds a very small amount of overhead to achieve 100% testability of all the modules and determines the initial resulting testing time, T_{init}. Our controllability enhancement technique chooses one module at a time to enhance so as to improve controllability of a number of others. Similarly,

observability of one module is normally enhanced to improve observability of a number of others.

Later, the design is modified so that the use of BIST resources is optimized under the given testing time constraint. Our approach achieves this by shrinking or stretching the test schedule with respect to T_{init}. Stretching test schedule is done by removing the maximum amount of BIST resources in such a way that the resulting ones will still keep the design testable and testing time constraint will be satisfied. Shrinking test schedule is achieved by adding a minimal amount of additional BIST resources into the design so that the tighter testing time constraint will be satisfied. These two heuristics, complemented with the heuristics of test pattern sharing and test response redirection, optimize BIST hardware overhead under the given testing time constraint.

Experimental results reveal that satisfying very short testing time constraints is very expensive. An average of 100% additional BIST hardware overhead on top of that incurred by our optimization by test pattern sharing and test response redirection is needed to guarantee that all operations are tested in one test session. We have also observed that there exists a limit on the testing time beyond which relaxing testing time constraints does not lead to any more hardware reduction.

In Chapter 5, we have defined a set of BIST transformations and proposed two approaches to solve the wiring-aware BIST synthesis problem.

The first approach uses a simulated annealing strategy to minimize the total design area. Experimental results indicate that when wiring area is considered during the optimization process the total area of the resulting designs is smaller compared to cases when wiring is not considered, despite the fact that the number of BIST registers may be smaller in the latter case. Since in deep-sub-micron technology wiring occupies a relatively large area, it is necessary to include wiring consideration early in the design processes.

The simulated annealing based optimization approach is, however, quite slow. To address this problem, the second approach, a greedy heuristic, has been proposed. It provides two ways to

converge towards testable and cheap solution while keeping computational effort low. It minimizes the overall number of testability enhancements done on the design. This is assisted by our novel BIST enhancement metrics which are used to guide the synthesis process in such a way that each controllability or observability enhancement targets to improve as many modules as possible. This is complemented by a thorough local search of the cheapest solution for each enhancement performed. The cheapest alternative enhancement for a given module is used.

Experimental results show that the approach is one order of magnitude faster than the simulated annealing. With large designs simulated annealing runs extremely slow, whereas our greedy heuristic produces results in acceptable run time.

6.2 Future Work

This work can be extended in a number of ways. Some possible directions for future work are:

- Extend the wiring-aware BIST synthesis problem with testing time constraints.
- In this thesis we have seen that several alternative testability environment options (ATEO) exist for each SDFG operation. This work has used behavioral information in the SDFG for testability analysis and BIST enhancement was done on the already allocated RTL design. One way to improve this work can be to use the ATEOs as metrics to guide HLS for BIST. In this way BIST insertion can start earlier so that the allocation, and even scheduling, can be BIST-aware.
- The performance of our greedy heuristic can be improved by avoiding multiple invocation of the area cost computation for each possible testability enhancement for the given module. Extending the heuristic with an intelligent algorithm to compute the cost of the design after BIST transformation in an incremental fashion can achieve this.

References

[1] Abramovici, M., M. A. Breuer and A. D. Friedman, "Digital Systems Testing and Testable Designs", *IEEE Press*, 1995.

[2] Agrawal, V. D., C. R. Kime and K. K. Saluja, "A Tutorial on Built-In Self-Test Part 1: Principles", *IEEE Design & Test of Computers, Volume 10, Issue 1*, pp.73-82, 1993.

[3] Agrawal, V. D., C. R. Kime and K. K. Saluja, "A Tutorial on Built-In Self-Test Part 2 Applications", *IEEE Design & Test of Computers, Volume 10, Issue 2*, pp. 69-77, Jun. 1993.

[4] Agrawal, V. D., Lecture notes at: http://www.ece.wisc.edu/~va/COURSE/lectures.html.

[5] Alvandpour, A. and C. Svensson, "A Wire Capacitance Estimation Technique for Power Consuming Interconnections at High Levels of Abstraction", *Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS97)*, Louvain-la-Neuve, Belgium, 1997.

[6] Avra, L., "Allocation and Assignment in High-level Synthesis for Self-testable Data Paths", *Proceedings of International Test Conference*, pp.463-472, Nashville, 1991.

[7] Bai, P., C. Auth, S. Balakrishnan, M. Bost, R. Brain, V. Chikarmane, R. Heussner, M. Hussein, J. Hwang, D. Ingerly, R. James, J. Jeong, C. Kenyon, E. Lee, S-H. Lee, N. Lindert, M. Liu, Z. Ma, T. Marieb, A. Murthy, R. Nagisetty, S. Natarajan, J. Neirynck, A. Ott, C. Parker, J. Sebastian, R. Shaheed, S. Sivakumar, J. Steigerwald, S. Tyagi, C. Weber, B. Woolery, A. Yeoh, K. Zhang, and M. Bohr, "A 65nm Logic Technology Featuring 35nm Gate Lengths, Enhanced Channel Strain, 8 Cu Interconnect Layers, Low-k ILD and

0.57 μm^2 SRAM Cell", *IEEE International Electron Devices Meeting*, San Francisco, CA, U.S.A, Dec. 2004.

[8] Battacharya, S., F. Brglez and S. Dey, "Transformations and Resynthesis for Testability of RT level Control-Data Path Synthesis", *IEEE Transactions on VLSI Systems, Vol. 1*, pp. 304-318, Sep. 1993.

[9] Bhatia, S. and N. K. Jha, "Behavioral Synthesis for Hierarchical Testability of Controller/Data Path Circuits with Conditional Branches", *Proceedings of the International Conference on Computer Design*, pp.91-96, Boston, MA, U.S.A., Oct. 1994.

[10] Bhatia, S. and N. K. Jha, "Genesis: A Behavioral Synthesis System for Hierarchical Testability", *Proceedings of the European Design and Test Conference*, pp. 272-276, Feb. 1994.

[11] Boubezari, S., E. Cerny, B. Kaminska and B. Nadeau-Dostie, "Testability Analysis and Test-Point Insertion in RTL VHDL Specifications for Scan-Based BIST", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, Issue 9*, pp.1327–1340, Sept. 1999.

[12] Cha, D. W., W. E. Donath and F. Ozguner, "9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits", *IEEE Transactions on Computers*, Vol. C-27, pp. 193-200, March 1978.

[13] Chen, C. H., and P. R. Menon, "An Approach to Functional-level Testability Analysis", *Proceedings of the International Test Conference*, pp. 373-380, Washington, U.S.A., 1989.

[14] Chen, C.-I. H., and J. T. Yuen, "Concurrent Test Scheduling in Built-In Self-Test Environment", *Proceedings of IEEE VLSI International Conference in Computers Design (ICCD'92)*, pp.256 –259, Cambridge, MA U.S.A., Oct. 1992.

[15] Chiu, S. S. K., and C. A. Papachristou, "A Built-in Self-Testing Approach for Minimizing Hardware Overhead", *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '91)*, pp. 282-285, Cambridge, MA U.S.A., 14-16 Oct. 1991.

[16] Craig, G. L., C. R. Kime and K. Saluja, "Test Scheduling and Control for VLSI Built-In Self-Test", *IEEE Transactions on Computers, Vol. 37, Issue 9*, pp. 1099-1109, Sept. 1988.

[17] Dey, S., M. Potkonjak and R. K. Roy, "Exploiting Hardware Sharing in High-level Synthesis for Partial Scan Optimization", *Proceedings of the International Conference on Computer Design*, pp. 20-25, Santa Clara, CA U.S.A., Nov. 1993.

[18] Eles, P., Z. Peng and A. Doboli, "VHDL System-Level Specification and Partitioning in a Hardware/Software Co-Synthesis Environment", *Proceedings of the 3rd International Workshop on Hardware/Software Codesign*, 1994.

[19] Flottes, M. L., R. Pires and B. Rouzeyre, "Analyzing Testability from Behavioral to RT Level", *Proceedings of the European Design and Test Conference*, pp.158-165, Paris, March, 1997.

[20] Gajski, D., R. Dömer and J. Zhu, "IP-Centric Methodology and Design with the SpecC Language System Level Design of Embedded Systems", *In System Level Synthesis, Kluwer Academic Publishers*, 1999.

[21] Ghosh, I., A. Raghunathan and N. K. Jha, "Design for Hierarchical Testability of RTL Circuits Obtained by Behavioural Synthesis", *Proceedings of the International Conference on Computer Design*, pp. 173-179, Austin, TX, U.S.A., Oct. 1995.

[22] Ghosh, I., N. K. Jha and S. Bhawmik, "A BIST Scheme for RTL Circuits Based on Symbolic Testability Analysis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 19, Issue 1*, pp. 111–128, Jan. 2000.

[23] Ghosh, I., N.K. Jha and S. Bhawmik, "A BIST Scheme for RTL Controller-Data Paths Based on Symbolic Testability Analysis", *Proceedings of the 35th ACM IEEE Design Automation Conference*, pp. 554-559, San Francisco, CA, U.S.A, 1998.

[24] Goel, P., "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits", *IEEE Transaction on Computers, Vol. C-30, No. 3, pp. 215-222.*

[25] Goel, S. K., and E. J. Marinissen, "Layout-Driven SoC Test Architecture Design for Test Time and Wire Length Minimization", *Proceedings of Design, Automation and Test in Europe*, pp.738-743, 2003.

[26] Gu, X., "RT-Level Testability Driven Partitioning", *Proceedings of the 13th IEEE VLSI Test Symposium*, pp.176-81, Princeton U.S.A., May 1995.

[27] Gu, X., K. Kuchcinski and Z. Peng, "An Efficient and Economic Partitioning Approach for Testability", *Proceedings of the International Test Conference*, pp.403-412, Washington D.C., U.S.A., 1995.

[28] Gu, X., K. Kuchcinski, Z. Peng, "Testability Analysis and Improvement from VHDL Behavioral Specifications", *Proceedings of the European Design Automation Conference*, pp. 644-649, Sept. 1994, Grenoble, France.

[29] Hallberg, J. and Z. Peng, "Estimation and Consideration of Interconnection Delays during High Level Synthesis", *Proceedings of the 24th Euromicro Conference, Vol. 1*, pp.349-356, Vasteras Sweden, Aug. 1998.

[30] Harmanani, H., C. Papachristou, S. Chiu and M. Nourani, "SYNTEST: An Environment for System-level Design for Test", *Proceedings of the European Design Automation Conference (EURO-DAC'92)*, pp. 402-407, Hamburg, Germany, Sep.1992.

[31] Harris, I. G. and A. Orailoglu, "SYNCBIST: Synthesis for Concurrent Built-in Self-Testability", Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors (ICCD'94), pp.101-104, Cambridge, MA, U.S.A., Oct. 1994.

[32] Harris, I. G., A. Orailoglu, "Fine-Grained Concurrency in Test Scheduling for Partial-Intrusion BIST", *Proceedings of European Design and Test Conference*, pp. 119-123, Paris, France, 1994.

[33] Harris, I. G., and A. Orailoglu, "Microarchitectural Synthesis of VLSI Designs with High Test Concurrency", *Proceedings of the IEEE Design Automation Conference*, pp.206-211, San Diego, CA U.S.A., June 1994.

[34] Howes, M. J., and D. V. Morgan, Reliability and Degradation-Semiconductor Devices and Circuits, *John Wiley & Sons Inc.*, 1981.

[35] Hwang, E. O., "Digital Logic and Microprocessor Design with VHDL", Thomson-Engineering, 2005.

[36] Jerraya, A. A., M. Romdhani, Ph. Le Marrec, F. Hessel, P. Coste, C. Valderrama, G. F. Marchioro, J. M. Daveau, N. –E. Zergainoh, "Multilanguage Specification for System Design", *In System Level Synthesis, Kluwer Academic Publishers*, 1999.

[37] Jervan, G., "High-Level Test Generation and Built-In Self-test Techniques for Digital Systems", Licenciate Thesis, *Linköping University*, 2002.

[38] Jervan, G., Z. Peng, R. Ubar and H. Kruus, "A Hybrid BIST Architecture and its Optimization for SoC Testing", *Proceedings of the IEEE 2002 3rd International Symposium on Quality Electronic Design (ISQED'02)*, pp. 273-279, San Jose, California, USA, March 18-20, 2002.

[39] Kim, H. B., T. Takahashi and D. S. Ha, "Test Session Built-In Self-testable Data Path Synthesis", Proceedings of International Test Conference, pp.154–163, Washington D.C. U.S.A., 1998.

[40] Kim, T., K. -S. Chung and C. L. Liu, "A Stepwise Refinement Data-path Synthesis Procedure for Easy Testability", *Proceedings of the European Design and Test Conference*, pp.586-590, Paris, France, 1994.

[41] Larsson, E. and Z. Peng, "Testability Analysis of Behavioral-Level Specifications", *Proceedings of the European Test Workshop*, pp.143-144, Sitges, May 1998.

[42] Larsson, E., "High-Level Testability Analysis and Enhancement Techniques", Licenciate Thesis no. 725, *Linköping University*, 1998.

[43] Lavagno, L., A. Sangiovanni-Vincentelli and E. Senyovich, "Models of Computation for Embedded System Design", *In System Level Synthesis, Kluwer Academic Publishers*, 1999.

[44] Lee, M. T-C., "High-Level Test Synthesis of Digital VLSI Circuits", *Artech House*, 1997.

[45] Lee, T. C., W. H. Wolf and N. K. Jha, "Behavioral Synthesis of easy Testability in Data Paths Scheduling", *Proceedings of the International Conference on Computer Design*, pp.616-619, Nov. 1992.

[46] Lee, T. C., W. H. Wolf, J. M. Acken and N. K. Jha, "Behavioral Synthesis of Easy Testability in Data Paths Allocation", *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors (ICCD'92)*, pp.29-32, 1992.

[47] Li, X. and P. Y. S. Cheung, "Exploiting Test Resource Optimization in Data Path Synthesis for BIST", *Proceedings of 9th Great Lakes Symposium on VLSI*, pp.342 –343, Ypsilanti, MI, U.S.A., March 1999.

[48] Micheli, G. D., "Synthesis and Optimization of Digital Circuits", *McGraw-Hill*, 1994.

[49] Mohamed, A. R., Z. Peng and P. Eles, "A Wiring-Aware Approach to Minimizing Built-In Self-Test Overhead", *Proceedings of the IEEE International Workshop on Electronic Design, Test and Applications (DELTA 2004)*, pp.413-415, Perth, Australia, Jan. 28-30, 2004.

[50] Mohamed, A. R., Z. Peng and P. Eles, "BIST Synthesis: An Approach to Resources Optimization under Test Time Constraints", *Proceedings of the 5th Design and Diagnostic of Electronic Circuits and Systems (DDECS2002)*, Brno, Czech Republic, pp. 346-351, 2002.

[51] Moshnyaga, V. G., and K. Tumaru, "A Placement Driven Methodology for High-Level Synthesis of Sub-Micron ASICS", *International Symposium on Circuits and Systems (ISCAS'96)*, May 1996, pp. 572-575.

[52] Mourad, S. and Y. Zorian, "Principles of Testing Electronic Systems", *John Wiley & Sons Inc.*, 2000.

[53] Nicolici, N. and B. M. Al-Hashimi, "Efficient BIST Hardware Insertion with Low Test Application Time for Synthesized Data Paths", Proceedings of Design, Automation and Test In Europe, pp.289-295, Munich, Germany, March 1999.

[54] Nicolici, N., B. M. Al-Hashimi A. D. Brown and A. C. Williams, "BIST Hardware Synthesis for RTL Data Paths based on Test Compatibility Classes", IEEE Transactions on

Computer-Aided Design of Integrated Circuits and Systems, Vol. 19, Issue 11, pp.1375–1385, Nov 2000.

[55] Papachristou, C. A., M. Baklashov and K. Lai, "High-Level Test Synthesis of Behavioral and Structural Designs", *Journal of Electronic Testing and Applications (JETTA), Vol. 13, No. 2*, pp.167-188, 1998.

[56] Peng, Z. and K. Kuchcinski, "Automated Transformation of Algorithms into Register-Transfer Level Implementations", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 13*, pp.150-166, 1994.

[57] Potkonjak, M., S. Dey and R. K. Roy, "Behavioral Synthesis of Area-Efficient Testable Designs Using Interaction between Hardware Sharing and Partial Scan", *IEEE Transaction on Computer-Aided Design, Vol. 14, No. 9*, pp.1141-1154, Sep. 1995.

[58] Ravi, S., G. Lakshminarayana and N. K. Jha, "TAO: Regular Expression based High-Level Testability Analysis and Optimization", *Proceedings of the International Test Conference (ITC)*, pp. 331–340, Washington D.C. U.S.A., Oct.1998.

[59] Ravi, S., N. K. Jha and G. Lakshminarayana, "TAO-BIST: A Framework for Testability Analysis and Optimization of RTL Circuits for BIST", *Proceedings of the 17th IEEE VLSI Test Symposium*, pp. 398-406, Dana Point, CA, U.S.A., 1999.

[60] Reeves, C. R., "Modern Heuristic Techniques for Combinatorial Problems", *Blackwell Scientific Publications*, 1993.

[61] Roth, J. P., W. G. Bouricius, and P. R. Schneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish between Failures in Logic Circuits", *IEEE Transactions on Computers, Vol. EC-16, No. 10, pp. 567-580.*

[62] Semiconductor Industry Association, "The International Technology Roadmap for Semiconductors (ITRS2001)", *http://public.itrs.net/*, 2001.

[63] Sugihara, M., H. Date and H. Yasuura, "Analysis and Minimization of Test Time in a combined BIST and External Test Approach", *Proceedings of Design Automation and Test in Europe*, pp.134–140, Paris, France, 2000.

[64] Vahidi, M., and A. Orailoglu, “Metric-Based Transformations for Self-testable VLSI Designs with High test Concurrency”, *Proceedings of EURO-DAC '95*, pp.136-141, 1995.

[65] Vahidi, M., and A. Orailoglu, “Testability Metrics for Synthesis of Self Testable Designs and Effective Test Plans", *Proceedings of 13th IEEE VLSI Test Symposium*, pp.170 –175, 1995.

[66] Wolf, W., “Hardware/Software Co-Synthesis Algorithms”, *In System Level Synthesis, Kluwer Academic Publishers*, 1999.

[67] Yang, T., and Z. Peng, “An Efficient Algorithm to Integrate Scheduling and Allocation in High-level Test Synthesis”, *Proceedings of the Design, Automation and Test in Europe*, pp.74-81, Paris, France, Feb. 1998.

[68] Zorian, Y., S. Dey and M. J. Rodgers, “Test of Future System-on-Chips”, *Proceedings of the IEE/ACM International Conference on Computer Aided Design (ICCAD-2000)*, pp. 392 – 398, 5-9 Nov. 2000.

List of Acronyms

ATE	Automatic Test Equipment
ATEO	Alternative Test Environment Option
ATPG	Automatic Test Pattern Generation
BDD	Binary Decision Diagram
BILBO	Built In Logic Block Observer
BIST	Built-in Self Test
CBILBO	Concurrent Built In Logic Block Observer
CUT	Circuit under Test
DFG	Data Flow Graph
DfT	Design for Testability
ECEP	Effective Controllability Enhancement Potential
EOEP	Effective Observability Enhancement Potential
ETPN	Extended Timed Petri Net
FC	Fault Coverage
HLS	High-level Synthesis
HLTS	High-level Test Synthesis
ILP	Integer Linear Programming
IP	Intellectual Property
LFSR	Linear Feedback Shift Register

MISR	Multiple Input Signature Register
MISRIS	MISR Incompatibility Set
MNACRM	Maximum number of allowed consecutive rejected moves
MUT	Module under Test
NoC	Network-on-Chip
RTL	Register-transfer Level
SA	Simulated Annealing
s-a-v	Stuck at a value *v*
SDFG	Scheduled Data Flow Graph
SoC	System-on-Chip
STA	Symbolic Testability Analysis
TCDF	Test Control Data Flow
TCEP	Total Controllability Enhancement Potential
TE	Test Environment
TOEP	Total Observability Enhancement Potential
TPG	Test Pattern Generator
VHDL	VLSI High Description Language or Very high speed integrated circuit Hardware Description Language
VLSI	Very Large Scale Integration

LINKÖPINGS UNIVERSITET

Avdelning, institution
Division, department

Institutionen för datavetenskap

Department of Computer and Information Science

Datum
Date

2005-04-28

Språk
Language

[ ] Svenska/Swedish
[X] Engelska/English
[ ] _____

Rapporttyp
Report category

[X] Licentiatavhandling
[ ] Examensarbete
[ ] C-uppsats
[ ] D-uppsats
[ ] Övrig rapport

_____

URL för elektronisk version

http://www.ida.liu.se/~eslab

ISBN 91-85297-90-9

ISRN LiU-Tek-Lic-2005:11

Serietitel och serienummer
Title of series, numbering

ISSN 0280-7971

Linköping Studies in Science and Technology

Thesis No. 1156

Titel
Title

High-Level Techniques for Built-In Self-Test Resources Optimization

Författare
Author

Abdil Rashid Mohamed

Sammanfattning
Abstract

Design modifications to improve testability usually introduce large area overhead and performance degradation. One way to reduce the negative impact associated with improved testability is to take testability as one of the constraints during high-level design phases so that systems are not only optimized for area and performance, but also from the testability point of view. This thesis deals with the problem of optimizing testing-hardware resources by taking into account testability constraints at high-levels of abstraction during the design process.

Firstly, we have provided an approach to solve the problem of optimizing built-in self-test (BIST) resources at the behavioral and register-transfer levels under testability and testing time constraints. Testing problem identification and BIST enhancement during the optimization process are assisted by symbolic testability analysis. Further, concurrent test sessions are generated, while signature analysis registers' sharing conflicts as well as controllability and observability constraints are considered.

Secondly, we have introduced the problem of BIST resources insertion and optimization while taking wiring area into account. Testability improvement transformations have been defined and deployed in a hardware overhead minimization technique used during a BIST synthesis process. The technique is guided by the results of symbolic testability analysis and inserts a minimal amount of BIST resources into the design to make it fully testable. It takes into consideration both BIST components cost and wiring overhead. Two design space exploration approaches have been proposed: a simulated annealing based algorithm and a greedy heuristic. Experimental results show that considering wiring area during BIST synthesis results in smaller final designs as compared to the cases when the wiring impact is ignored. The greedy heuristic uses our behavioral and register-transfer levels BIST enhancement metrics to guide BIST synthesis in such a way that the number of testability improvement transformations performed on the design is reduced.

The Swedish Foundation for Strategic Research (SSF) under the INTELECT and STRINGENT programmes at Linköping University supported this work.

Nyckelord
Keywords

Built-in Self-test, Test Synthesis, Testability Analysis, Wiring-aware BIST Synthesis

Department of Computer and Information Science
Linköpings universitet

Linköping Studies in Science and Technology
Faculty of Arts and Sciences - Licentiate Theses

No 17 **Vojin Plavsic:** Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)

No 28 **Arne Jönsson, Mikael Patel:** An Interactive Flowcharting Technique for Communicating and Realizing Algorithms, 1984.

No 29 **Johnny Eckerland:** Retargeting of an Incremental Code Generator, 1984.

No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching, 1985.

No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems, 1985.

No 60 **Johan Fagerström:** Simulation and Evaluation of Architecture based on Asynchronous Processes, 1985.

No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System, 1987.

No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems, 1986.

No 73 **Ola Strömfors:** A Structure Editor for Documents and Programs, 1986.

No 74 **Christos Levcopoulos:** New Results about the Approximation Behavior of the Greedy Triangulation, 1986.

No 104 **Shamsul I. Chowdhury: S**tatistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology, 1987.

No 108 **Rober Bilos:** Incremental Scanning and Token-Based Editing, 1987.

No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments, 1987.

No 113 **Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge, 1987.

No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs, 1987.

No 126 **Dan Strömberg:** Transfer and Distribution of Application Programs, 1987.

No 127 **Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems, 1987.

No 139 **Christer Bäckström:** Reasoning about Interdependent Actions, 1988.

No 140 **Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing, 1988.

No 146 **Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System, 1988.

No 150 **Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behavior, 1988.

No 165 **Jonas Löwgren:** Supporting Design and Management of Expert System User Interfaces, 1989.

No 166 **Ola Petersson:** On Adaptive Sorting in Sequential and Parallel Models, 1989.

No 174 **Yngve Larsson:** Dynamic Configuration in a Distributed Environment, 1989.

No 177 **Peter Åberg:** Design of a Multiple View Presentation and Interaction Manager, 1989.

No 181 **Henrik Eriksson:** A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.

No 184 **Ivan Rankin:** The Deep Generation of Text in Expert Critiquing Systems, 1989.

No 187 **Simin Nadjm-Tehrani:** Contributions to the Declarative Approach to Debugging Prolog Programs, 1989.

No 189 **Magnus Merkel:** Temporal Information in Natural Language, 1989.

No 196 **Ulf Nilsson:** A Systematic Approach to Abstract Interpretation of Logic Programs, 1989.

No 197 **Staffan Bonnier:** Horn Clause Logic with External Procedures: Towards a Theoretical Framework, 1989.

No 203 **Christer Hansson:** A Prototype System for Logical Reasoning about Time and Action, 1990.

No 212 **Björn Fjellborg:** An Approach to Extraction of Pipeline Structures for VLSI High-Level Synthesis, 1990.

No 230 **Patrick Doherty:** A Three-Valued Approach to Non-Monotonic Reasoning, 1990.

No 237 **Tomas Sokolnicki:** Coaching Partial Plans: An Approach to Knowledge-Based Tutoring, 1990.

No 250 **Lars Strömberg:** Postmortem Debugging of Distributed Systems, 1990.

No 253 **Torbjörn Näslund:** SLDFA-Resolution - Computing Answers for Negative Queries, 1990.

No 260 **Peter D. Holmes:** Using Connectivity Graphs to Support Map-Related Reasoning, 1991.

No 283 **Olof Johansson:** Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge-Bases, 1991.

No 298 **Rolf G Larsson:** Aktivitetsbaserad kalkylering i ett nytt ekonomisystem, 1991.

No 318 **Lena Srömbäck:** Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems, 1992.

No 319 **Mikael Pettersson:** DML-A Language and System for the Generation of Efficient Compilers from Denotational Specification, 1992.

No 326 **Andreas Kågedal:** Logic Programming with External Procedures: an Implementation, 1992.

No 328 **Patrick Lambrix:** Aspects of Version Management of Composite Objects, 1992.

No 333 **Xinli Gu:** Testability Analysis and Improvement in High-Level Synthesis Systems, 1992.

No 335 **Torbjörn Näslund:** On the Role of Evaluations in Iterative Development of Managerial Support Sytems, 1992.

No 348 **Ulf Cederling:** Industrial Software Development - a Case Study, 1992.

No 352 **Magnus Morin:** Predictable Cyclic Computations in Autonomous Systems: A Computational Model and Implementation, 1992.

No 371 **Mehran Noghabai:** Evaluation of Strategic Investments in Information Technology, 1993.

No 378 **Mats Larsson:** A Transformational Approach to Formal Digital System Design, 1993.

No 380 **Johan Ringström:** Compiler Generation for Parallel Languages from Denotational Specifications, 1993.

No 381 **Michael Jansson:** Propagation of Change in an Intelligent Information System, 1993.

No 383 **Jonni Harrius:** An Architecture and a Knowledge Representation Model for Expert Critiquing Systems, 1993.

No 386 **Per Österling:** Symbolic Modelling of the Dynamic Environments of Autonomous Agents, 1993.

No 398 **Johan Boye:** Dependency-based Groudness Analysis of Functional Logic Programs, 1993.

No 402 **Lars Degerstedt:** Tabulated Resolution for Well Founded Semantics, 1993.
No 406 **Anna Moberg:** Satellitkontor - en studie av kommunikationsmönster vid arbete på distans, 1993.
No 414 **Peter Carlsson:** Separation av företagsledning och finansiering - fallstudier av företagsledarutköp ur ett agentteoretiskt perspektiv, 1994.
No 417 **Camilla Sjöström:** Revision och lagreglering - ett historiskt perspektiv, 1994.
No 436 **Cecilia Sjöberg:** Voices in Design: Argumentation in Participatory Development, 1994.
No 437 **Lars Viklund:** Contributions to a High-level Programming Environment for a Scientific Computing, 1994.
No 440 **Peter Loborg:** Error Recovery Support in Manufacturing Control Systems, 1994.
FHS 3/94 **Owen Eriksson:** Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och samskapande perspektiv, 1994.
FHS 4/94 **Karin Pettersson:** Informationssystemstrukturering, ansvarsfördelning och användarinflytande - En komparativ studie med utgångspunkt i två informationssystemstrategier, 1994.
No 441 **Lars Poignant:** Informationsteknologi och företagsetablering - Effekter på produktivitet och region, 1994.
No 446 **Gustav Fahl:** Object Views of Relational Data in Multidatabase Systems, 1994.
No 450 **Henrik Nilsson:** A Declarative Approach to Debugging for Lazy Functional Languages, 1994.
No 451 **Jonas Lind:** Creditor - Firm Relations: an Interdisciplinary Analysis, 1994.
No 452 **Martin Sköld:** Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, 1994.
No 455 **Pär Carlshamre:** A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Systems Development, 1994.
FHS 5/94 **Stefan Cronholm:** Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer, 1994.
No 462 **Mikael Lindvall:** A Study of Traceability in Object-Oriented Systems Development, 1994.
No 463 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av Sandviks förvärv av Bahco Verktyg, 1994.
No 464 **Hans Olsén:** Collage Induction: Proving Properties of Logic Programs by Program Synthesis, 1994.
No 469 **Lars Karlsson:** Specification and Synthesis of Plans Using the Features and Fluents Framework, 1995.
No 473 **Ulf Söderman:** On Conceptual Modelling of Mode Switching Systems, 1995.
No 475 **Choong-ho Yi:** Reasoning about Concurrent Actions in the Trajectory Semantics, 1995.
No 476 **Bo Lagerström:** Successiv resultatavräkning av pågående arbeten. - Fallstudier i tre byggföretag, 1995.
No 478 **Peter Jonsson:** Complexity of State-Variable Planning under Structural Restrictions, 1995.
FHS 7/95 **Anders Avdic:** Arbetsintegrerad systemutveckling med kalkylkprogram, 1995.
No 482 **Eva L Ragnemalm:** Towards Student Modelling through Collaborative Dialogue with a Learning Companion, 1995.
No 488 **Eva Toller:** Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
No 489 **Erik Stoy:** A Petri Net Based Unified Representation for Hardware/Software Co-Design, 1995.
No 497 **Johan Herber:** Environment Support for Building Structured Mathematical Models, 1995.
No 498 **Stefan Svenberg:** Structure-Driven Derivation of Inter-Lingual Functor-Argument Trees for Multi-Lingual Generation, 1995.
No 503 **Hee-Cheol Kim:** Prediction and Postdiction under Uncertainty, 1995.
FHS 8/95 **Dan Fristedt:** Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
FHS 9/95 **Malin Bergvall:** Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och ansvarsroller, 1995.
No 513 **Joachim Karlsson:** Towards a Strategy for Software Requirements Selection, 1995.
No 517 **Jakob Axelsson:** Schedulability-Driven Partitioning of Heterogeneous Real-Time Systems, 1995.
No 518 **Göran Forslund:** Toward Cooperative Advice-Giving Systems: The Expert Systems Experience, 1995.
No 522 **Jörgen Andersson:** Bilder av småföretagares ekonomistyrning, 1995.
No 538 **Staffan Flodin:** Efficient Management of Object-Oriented Queries with Late Binding, 1996.
No 545 **Vadim Engelson:** An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing, 1996.
No 546 **Magnus Werner :** Multidatabase Integration using Polymorphic Queries and Views, 1996.
FiF-a 1/96 **Mikael Lind:** Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
No 549 **Jonas Hallberg:** High-Level Synthesis under Local Timing Constraints, 1996.
No 550 **Kristina Larsen:** Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag. 1996.
No 557 **Mikael Johansson:** Quality Functions for Requirements Engineering Methods, 1996.
No 558 **Patrik Nordling:** The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
No 561 **Anders Ekman:** Exploration of Polygonal Environments, 1996.
No 563 **Niclas Andersson:** Compilation of Mathematical Models to Parallel Code, 1996.
No 567 **Johan Jenvald:** Simulation and Data Collection in Battle Training, 1996.
No 575 **Niclas Ohlsson:** Software Quality Engineering by Early Identification of Fault-Prone Modules, 1996.
No 576 **Mikael Ericsson:** Commenting Systems as Design Support—A Wizard-of-Oz Study, 1996.
No 587 **Jörgen Lindström:** Chefers användning av kommunikationsteknik, 1996.
No 589 **Esa Falkenroth:** Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
No 591 **Niclas Wahllöf:** A Default Extension to Description Logics and its Applications, 1996.
No 595 **Annika Larsson:** Ekonomisk Styrning och Organisatorisk Passion - ett interaktivt perspektiv, 1997.
No 597 **Ling Lin:** A Value-based Indexing Technique for Time Sequences, 1997.

No 598 **Rego Granlund:** C^3Fire - A Microworld Supporting Emergency Management Training, 1997.
No 599 **Peter Ingels:** A Robust Text Processing Technique Applied to Lexical Error Recovery, 1997.
No 607 **Per-Arne Persson:** Toward a Grounded Theory for Support of Command and Control in Military Coalitions, 1997.
No 609 **Jonas S Karlsson:** A Scalable Data Structure for a Parallel Data Server, 1997.
FiF-a 4 **Carita Åbom:** Videomötesteknik i olika affärssituationer - möjligheter och hinder, 1997.
FiF-a 6 **Tommy Wedlund**: Att skapa en företagsanpassad systemutvecklingsmodell - genom rekonstruktion, värdering och vidareutveckling i T50-bolag inom ABB, 1997.
No 615 **Silvia Coradeschi**: A Decision-Mechanism for Reactive and Coordinated Agents, 1997.
No 623 **Jan Ollinen:** Det flexibla kontorets utveckling på Digital - Ett stöd för multiflex? 1997.
No 626 **David Byers:** Towards Estimating Software Testability Using Static Analysis, 1997.
No 627 **Fredrik Eklund:** Declarative Error Diagnosis of GAPLog Programs, 1997.
No 629 **Gunilla Ivefors:** Krigsspel coh Informationsteknik inför en oförutsägbar framtid, 1997**.**
No 631 **Jens-Olof Lindh:** Analysing Traffic Safety from a Case-Based Reasoning Perspective, 1997
No 639 **Jukka Mäki-Turja:**. Smalltalk - a suitable Real-Time Language, 1997.
No 640 **Juha Takkinen**: CAFE: Towards a Conceptual Model for Information Management in Electronic Mail, 1997.
No 643 **Man Lin**: Formal Analysis of Reactive Rule-based Programs, 1997.
No 653 **Mats Gustafsson**: Bringing Role-Based Access Control to Distributed Systems, 1997.
FiF-a 13 **Boris Karlsson:** Metodanalys för förståelse och utveckling av systemutvecklingsverksamhet. Analys och värdering av systemutvecklingsmodeller och dess användning, 1997.
No 674 **Marcus Bjäreland:** Two Aspects of Automating Logics of Action and Change - Regression and Tractability, 1998.
No 676 **Jan Håkegård**: Hiera rchical Test Architecture and Board-Level Test Controller Synthesis, 1998.
No 668 **Per-Ove Zetterlund**: Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998.
No 675 **Jimmy Tjäder**: Projektledaren & planen - en studie av projektledning i tre installations- och systemutvecklingsprojekt, 1998.
FiF-a 14 **Ulf Melin**: Informationssystem vid ökad affärs- och processorientering - egenskaper, strategier och utveckling, 1998.
No 695 **Tim Heyer**: COMPASS: Introduction of Formal Methods in Code Development and Inspection, 1998.
No 700 **Patrik Hägglund:** Programming Languages for Computer Algebra, 1998.
FiF-a 16 **Marie-Therese Christiansson:** Inter-organistorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998.
No 712 **Christina Wennestam:** Information om immateriella resurser. Investeringar i forskning och utveckling samt i personal inom skogsindustrin, 1998.
No 719 **Joakim Gustafsson:** Extending Temporal Action Logic for Ramification and Concurrency, 1998.
No 723 **Henrik André-Jönsson:** Indexing time-series data using text indexing methods, 1999.
No 725 **Erik Larsson:** High-Level Testability Analysis and Enhancement Techniques, 1998.
No 730 **Carl-Johan Westin:** Informationsförsörjning: en fråga om ansvar - aktiviteter och uppdrag i fem stora svenska organisationers operativa informationsförsörjning, 1998.
No 731 **Åse Jansson:** Miljöhänsyn - en del i företags styrning, 1998.
No 733 **Thomas Padron-McCarthy:** Performance-Polymorphic Declarative Queries, 1998.
No 734 **Anders Bäckström:** Värdeskapande kreditgivning - Kreditriskhantering ur ett agentteoretiskt perspektiv, 1998.
FiF-a 21 **Ulf Seigerroth:** Integration av förändringsmetoder - en modell för välgrundad metodintegration, 1999.
FiF-a 22 **Fredrik Öberg:** Object-Oriented Frameworks - A New Strategy for Case Tool Development, 1998.
No 737 **Jonas Mellin:** Predictable Event Monitoring, 1998.
No 738 **Joakim Eriksson:** Specifying and Managing Rules in an Active Real-Time Database System, 1998.
FiF-a 25 **Bengt E W Andersson:** Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarenor i samverkan om utbyte av information, 1998.
No 742 **Pawel Pietrzak:** Static Incorrectness Diagnosis of CLP (FD), 1999.
No 748 **Tobias Ritzau:** Real-Time Reference Counting in RT-Java, 1999.
No 751 **Anders Ferntoft:** Elektronisk affärskommunikation - kontaktkostnader och kontaktprocesser mellan kunder och leverantörer på producentmarknader,1999.
No 752 **Jo Skåmedal:** Arbete på distans och arbetsformens påverkan på resor och resmönster, 1999.
No 753 **Johan Alvehus:** Mötets metaforer. En studie av berättelser om möten, 1999.
No 754 **Magnus Lindahl:** Bankens villkor i låneavtal vid kreditgivning till högt belånade företagsförvärv: En studie ur ett agentteoretiskt perspektiv, 2000.
No 766 **Martin V. Howard:** Designing dynamic visualizations of temporal data, 1999.
No 769 **Jesper Andersson:** Towards Reactive Software Architectures, 1999.
No 775 **Anders Henriksson:** Unique kernel diagnosis, 1999.
FiF-a 30 **Pär J. Ågerfalk:** Pragmatization of Information Systems - A Theoretical and Methodological Outline, 1999.
No 787 **Charlotte Björkegren:** Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.
No 788 **Håkan Nilsson:** Informationsteknik som drivkraft i granskningsprocessen - En studie av fyra revisionsbyråer, 2000.
No 790 **Erik Berglund:** Use-Oriented Documentation in Software Development, 1999.
No 791 **Klas Gäre:** Verksamhetsförändringar i samband med IS-införande, 1999.
No 800 **Anders Subotic:** Software Quality Inspection, 1999.
No 807 **Svein Bergum**: Managerial communication in telework, 2000.

No 809 **Flavius Gruian:** Energy-Aware Design of Digital Systems, 2000.
FiF-a 32 **Karin Hedström:** Kunskapsanvändning och kunskapsutveckling hos verksamhetskonsulter - Erfarenheter från ett FOU-samarbete, 2000.
No 808 **Linda Askenäs:** Affärssystemet - En studie om teknikens aktiva och passiva roll i en organisation, 2000.
No 820 **Jean Paul Meynard:** Control of industrial robots through high-level task programming, 2000.
No 823 **Lars Hult:** Publika Gränsytor - ett designexempel, 2000.
No 832 **Paul Pop:** Scheduling and Communication Synthesis for Distributed Real-Time Systems, 2000.
FiF-a 34 **Göran Hultgren:** Nätverksinriktad Förändringsanalys - perspektiv och metoder som stöd för förståelse och utveckling av affärsrelationer och informationssystem, 2000.
No 842 **Magnus Kald:** The role of management control systems in strategic business units, 2000.
No 844 **Mikael Cäker:** Vad kostar kunden? Modeller för intern redovisning, 2000.
FiF-a 37 **Ewa Braf**: Organisationers kunskapsverksamheter - en kritisk studie av ”knowledge management”, 2000.
FiF-a 40 **Henrik Lindberg:** Webbaserade affärsprocesser - Möjligheter och begränsningar, 2000.
FiF-a 41 **Benneth Christiansson:** Att komponentbasera informationssystem - Vad säger teori och praktik?, 2000.
No. 854 **Ola Pettersson:** Deliberation in a Mobile Robot, 2000.
No 863 **Dan Lawesson**: Towards Behavioral Model Fault Isolation for Object Oriented Control Systems, 2000.
No 881 **Johan Moe:** Execution Tracing of Large Distributed Systems, 2001.
No 882 **Yuxiao Zhao:** XML-based Frameworks for Internet Commerce and an Implementation of B2B e-procurement, 2001.
No 890 **Annika Flycht-Eriksson:** Domain Knowledge Management inInformation-providing Dialogue systems, 2001.
Fif-a 47 **Per-Arne Segerkvis**t: Webbaserade imaginära organisationers samverkansformer, 2001.
No 894 **Stefan Svarén:** Styrning av investeringar i divisionaliserade företag - Ett koncernperspektiv, 2001.
No 906 **Lin Han**: Secure and Scalable E-Service Software Delivery, 2001.
No 917 **Emma Hansson:** Optionsprogram för anställda - en studie av svenska börsföretag, 2001.
No 916 **Susanne Odar:** IT som stöd för strategiska beslut, en studie av datorimplementerade modeller av verksamhet som stöd för beslut om anskaffning av JAS 1982, 2002.
Fif-a-49 **Stefan Holgersson:** IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001.
Fif-a-51 **Per Oscarsson:**Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2001.
No 919 **Luis Alejandro Cortes:** A Petri Net Based Modeling and Verification Technique for Real-Time Embedded Systems, 2001.
No 915 **Niklas Sandell:** Redovisning i skuggan av en bankkris - Värdering av fastigheter. 2001.
No 931 **Fredrik Elg:** Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidens i kontroll av mikrovärlden Moro, 2002.
No 933 **Peter Aronsson:** Automatic Parallelization of Simulation Code from Equation Based Simulation Languages, 2002.
No 938 **Bourhane Kadmiry**: Fuzzy Control of Unmanned Helicopter, 2002.
No 942 **Patrik Haslum**: Prediction as a Knowledge Representation Problem: A Case Study in Model Design, 2002.
No 956 **Robert Sevenius:** On the instruments of governance - A law & economics study of capital instruments in limited liability companies, 2002.
FiF-a 58 **Johan Petersson:** Lokala elektroniska marknadsplatser - informationssystem för platsbundna affärer, 2002.
No 964 **Peter Bunus:** Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002.
No 973 **Gert Jervan:** High-Level Test Generation and Built-In Self-Test Techniques for Digital Systems, 2002.
No 958 **Fredrika Berglund:** Management Control and Strategy - a Case Study of Pharmaceutical Drug Development, 2002.
Fif-a 61 **Fredrik Karlsson:** Meta-Method for Method Configuration - A Rational Unified Process Case, 2002.
No 985 **Sorin Manolache:** Schedulability Analysis of Real-Time Systems with Stochastic Task Execution Times, 2002.
No 982 **Diana Szentiványi:** Performance and Availability Trade-offs in Fault-Tolerant Middleware, 2002.
No 989 **Iakov Nakhimovski:** Modeling and Simulation of Contacting Flexible Bodies in Multibody Systems, 2002.
No 990 **Levon Saldamli:** PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations, 2002.
No 991 **Almut Herzog:** Secure Execution Environment for Java Electronic Services, 2002.
No 999 **Jon Edvardsson:** Contributions to Program- and Specification-based Test Data Generation, 2002
No 1000 **Anders Arpteg:** Adaptive Semi-structured Information Extraction, 2002.
No 1001 **Andrzej Bednarski:** A Dynamic Programming Approach to Optimal Retargetable Code Generation for Irregular Architectures, 2002.
No 988 **Mattias Arvola:** Good to use! : Use quality of multi-user applications in the home, 2003.
FiF-a 62 **Lennart Ljung:** Utveckling av en projektivitetsmodell - om organisationers förmåga att tillämpa projektarbetsformen, 2003.
No 1003 **Pernilla Qvarfordt:** User experience of spoken feedback in multimodal interaction, 2003.
No 1005 **Alexander Siemers:** Visualization of Dynamic Multibody Simulation With Special Reference to Contacts, 2003.
No 1008 **Jens Gustavsson:** Towards Unanticipated Runtime Software Evolution, 2003.
No 1010 **Calin Curescu:** Adaptive QoS-aware Resource Allocation for Wireless Networks, 2003.
No 1015 **Anna Andersson:** Management Information Systems in Process-oriented Healthcare Organisations, 2003.
No 1018 **Björn Johansson:** Feedforward Control in Dynamic Situations, 2003.
No 1022 **Traian Pop:** Scheduling and Optimisation of Heterogeneous Time/Event-Triggered Distributed Embedded Systems, 2003.
FiF-a 65 **Britt-Marie Johansson:** Kundkommunikation på distans - en studie om kommunikationsmediets betydelse i affärstransaktioner, 2003.
No 1024 **Aleksandra Tesanovic:** Towards Aspectual Component-Based Real-Time System Development, 2003.

No 1034 **Arja Vainio-Larsson**: Designing for Use in a Future Context - Five Case Studies in Retrospect, 2003.
No 1033 **Peter Nilsson:** Svenska bankers redovisningsval vid reservering för befarade kreditförluster - En studie vid införandet av nya redovisningsregler, 2003.
Fif-a 69 **Fredrik Ericsson:** Information Technology for Learning and Acquiring of Work Knowledge, 2003.
No 1049 **Marcus Comstedt:** Towards Fine-Grained Binary Composition through Link Time Weaving, 2003.
No 1052 **Åsa Hedenskog:** Increasing the Automation of Radio Network Control, 2003.
No 1054 **Claudiu Duma:** Security and Efficiency Tradeoffs in Multicast Group Key Management, 2003.
Fif-a 71 **Emma Eliasson:** Effektanalys av IT-systems handlingsutrymme, 2003.
No 1055 **Carl Cederberg:** Experiments in Indirect Fault Injection with Open Source and Industrial Software, 2003.
No 1058 **Daniel Karlsson:** Towards Formal Verification in a Component-based Reuse Methodology, 2003.
FiF-a 73 **Anders Hjalmarsson:** Att etablera och vidmakthålla förbättringsverksamhet - behovet av koordination och interaktion vid förändring av systemutvecklingsverksamheter, 2004.
No 1079 **Pontus Johansson:** Design and Development of Recommender Dialogue Systems, 2004.
No 1084 **Charlotte Stoltz:** Calling for Call Centres - A Study of Call Centre Locations in a Swedish Rural Region, 2004.
FiF-a 74 **Björn Johansson:** Deciding on Using Application Service Provision in SMEs, 2004.
No 1094 **Genevieve Gorrell:** Language Modelling and Error Handling in Spoken Dialogue Systems, 2004.
No 1095 **Ulf Johansson:** Rule Extraction - the Key to Accurate and Comprehensible Data Mining Models, 2004.
No 1099 **Sonia Sangari:** Computational Models of Some Communicative Head Movements, 2004.
No 1110 **Hans Nässla:** Intra-Family Information Flow and Prospects for Communication Systems, 2004.
No 1116 **Henrik Sällberg:** On the value of customer loyalty programs - A study of point programs and switching costs, 2004.
FiF-a 77 **Ulf Larsson:** Designarbete i dialog - karaktärisering av interaktionen mellan användare och utvecklare i en systemutvecklingsprocess, 2004.
No 1126 **Andreas Borg:** Contribution to Management and Validation of Non-Functional Requirements, 2004.
No 1127 **Per-Ola Kristensson:** Large Vocabulary Shorthand Writing on Stylus Keyboard, 2004.
No 1132 **Pär-Anders Albinsson:** Interacting with Command and Control Systems: Tools for Operators and Designers, 2004.
No 1130 **Ioan Chisalita:** Safety-Oriented Communication in Mobile Networks for Vehicles, 2004.
No 1138 **Thomas Gustafsson:** Maintaining Data Consistency im Embedded Databases for Vehicular Systems, 2004.
No 1149 **Vaida Jakoniené:** A Study in Integrating Multiple Biological Data Sources, 2005.
No 1156 **Abdil Rashid Mohamed**: High-Level Techniques for Built-In Self-Test Resources Optimization, 2005.