IST-2000-29212

CoTest

COTEST *- New Techniques for Supporting Testability in Co-design Environments*

COTEST/D3

Report on Early DfT Support

Petru ELES, Gert JERVAN,
Abdil Rashid MOHAMMED, Zebo PENG

Linköping University
IDA/SaS/ESLAB
Linköping, Sweden

Contact person:
Zebo Peng Embedded Systems Laboratory Linköping University SE-58183 Linköping Sweden
Tel. +46 13 282067 Fax. +46 13 284499 E-mail: zpe@ida.liu.se

Related documents

- COTEST Technical Annex
- COTEST Report D1: “Report on benchmark identification and planning of experiments to be performed”
- COTEST Report D2: “Report on automatic generation of test benches from system-level descriptions”

Table of Contents

1. Introduction

Test has always been one of the most time and resource consuming tasks of the electronic system design cycle. Due to the increased complexity of such systems, traditional gate-level methods are not any more practical and more and more work has to be done at higher levels of abstraction.

An extremely important aspect, related to system testing, which has to be handled with priority in the early phases of the design process, is design for testability (DfT). In the early phases, system synthesis is performed starting from an implementation independent specification. Among the synthesis tasks at the system level are the selection of an efficient implementation architecture and also the partitioning of the specified functionality into components, which will be implemented by hardware and software respectively. It is very important that all these design tasks be performed with careful consideration of the overall testability of the resulting system.

The main goal of workpackage 3 of the COTEST project was assessment of the feasibility and evaluation of test-oriented system modifications. In the following we introduce some possible techniques and discuss their effectiveness for early DfT support.

2. Main Achievements

Our work has formed on the development of a framework to support reasoning about system testability and provide methods for DfT modifications in the early phases of the design cycle. The framework consists of the following tasks and is depicted in Figure 1:

- Hierarchical Test Generation at the Behavioral Level,
- High-Level BIST Insertion, and
- Hybrid BIST Architecture and Analysis.

Our approach starts from an implementation independent behavioral specification, which is analyzed by using a hierarchical test generation approach. This gives us an early estimate about testability of different modules in the specification. The obtained information is used in the high-level built-in self-test (BIST) insertion environment, which performs detailed analysis of previously identified hard to test modules and insertion of BIST resources into the design. The supported self-test architecture is a hybrid BIST architecture, which can be implemented either in hardware or in software. Our approach can find the right trade-off between two and this gives us the optimal solution in terms of test cost (test length and area overhead) without losing in test quality.

We have also implemented several key components of the proposed design environment to support DfT modifications in early phases of the design cycle. This environment will be discussed in detail in the chapter “Experimental Results”.

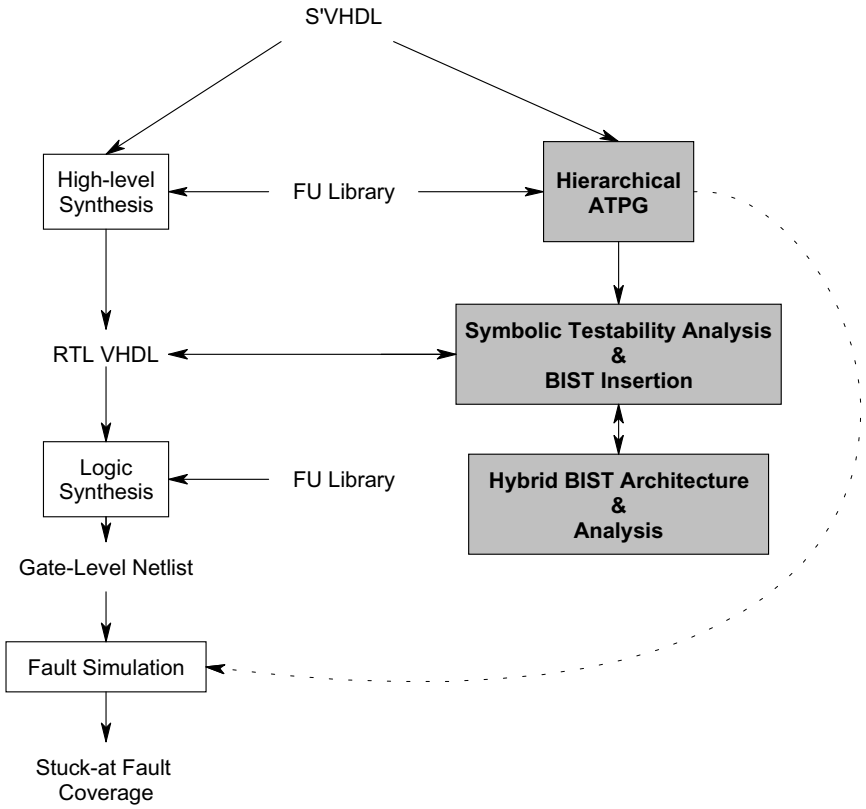


Figure 1: Framework for Early DfT Analysis and Modifications

3. Hierarchical Test Generation at the Behavioral Level

In our approach preliminary testability evaluation of the system is performed by an automatic test pattern generation (ATPG) algorithm and the obtained testability data is subsequently used for guiding DfT modifications. As discussed in COTEST report D2 [5], there exists a gap between the fault coverage figures attained by test sequences generated purely on a high-level and those by the gate-level ones. Therefore we have investigated possibilities of taking into account structural information during the test generation process and developed a novel high-level hierarchical test generation (HTG) algorithm. HTG is a technique which has been successfully used until now for hardware test generation at the gate, logical and register-transfer (RT) levels. Our HTG is employing constraint logic programming techniques and uses a decision diagram (DD) based representation. As a result of this project we demonstrate that HTG can be used successfully also at higher levels of abstraction. Another important advantage of our methodology is that, based on the divide and conquer strategy, it allows to generate tests for more complex systems, based on predesigned test vectors for the system modules. The test vectors for the individual modules can be generated based on different techniques suitable for the respective entities.

3.1. Behavioral modeling

3.1.1. S'VHDL

The design work starts with a behavioral specification. In our approach we are using for the subsequent synthesis step the CAMAD high-level synthesis tool [4], which is developed at Linköping University. It accepts as an input a behavioral specification specified in S'VHDL, which is the input format for our testability analysis and DfT toolflow.

S'VHDL [3] is defined as a subset of VHDL with the purpose of using it as input for high-level hardware synthesis. It is designed to accommodate a large behavioral subset of VHDL, particularly those constructs relevant for synthesis and to make available most of VHDL's facilities that support concurrency in the design.

3.1.2. Decision Diagrams

Decision Diagrams (DD) (previously known as Alternative Graphs) [18], [19] may represent a set of digital (Boolean or integer) functions *y=F(X)* corresponding to components or subcircuits in digital systems. Here, *y* is an output variable, and *X* is a vector of input variables of the component or subcircuit.

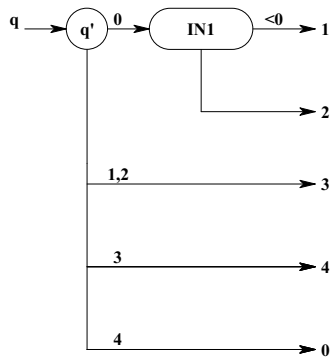
A DD describing digital systems on the behavioral level captures behavior instead of structure of the desired system. In DD, the variables in nonterminal nodes can be either Boolean (describing flags, logical conditions etc.) or integer (describing instruction words, control fields, etc.). The terminal nodes are labeled by constants, variables (Boolean or integer) or by expressions for calculating Boolean or integer values. The number of DDs, used for describing a digital system, is equal to the number of output and internal variables used in the instruction set description.

Figure 2 depicts an example of DD describing the behavior of a simple function. For example, variable *A* will be equal to *IN1+2*, if the system is in the state *q=2* (Figure 2c). If this state is to be activated, condition *IN1≥0* should be true *(*Figure 2b) and in our terminology this is a path activation constraint for activating a path to the specified state (*q=2*). The DDs, extracted from a specification, will

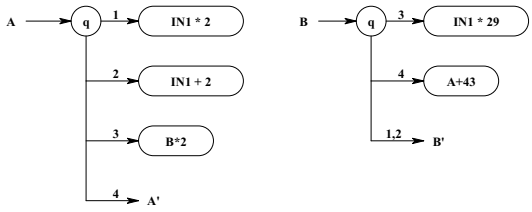
```
if (IN1 < 0) then
  A := IN1 * 2;  ------ q=1
else
  A := IN1 + 2; ------- q=2
endif;

B := IN1*29;    --------q=3
A := B*2;
B := A+43;     -------- q=4
```

a) Specification (comments start with "--")



b) Control-flow DD (q denotes the state variable and q' is the previous state)



c) Data-flow DD

Figure 2: A DD example

be used as a computational model in HTG for symbolic path activation.

3.2. Constraint Logic Programming

Most digital systems can be viewed as a set of constraints, which are a mathematical formalization of relationships that hold in the system [10]. In the view of test generation, there are two types of constraints: system constraints and test constraints. The system constraints describe the relationships between the system variables, which capture the system functionality and requirements. Test constraints describe the relationships between the system variables in order to generate tests for the system. Constraint solving can be viewed as a procedure to find a solution to satisfy the desired test constraints in a system if such a solution exists.

3.3. SICStus Prolog representation of Decision Diagrams

At the behavioral level there exist two types of DDs: control-flow DD and data-flow DDs. The control-flow DD carries two types of information: state transition information and path activation information. The state transition information captures the state transitions that are given in the FSM corresponding to the specified system. The path activation information holds conditions associated to state transitions.

For each internal or primary output variable corresponds one data-flow DD. In a certain system state, the value of a variable is determined by the terminal node in the data graph. In this case, the relationship between the terminal node and the variable can be viewed as a functional constraint on the variable at the state.

For solving different constraints we are employing a commercial constraint solver SICStus [15] and have developed a framework for representing a DD model in the form of constraints. First, we are translating the control-flow DD into a set of state transition predicates and path activation constraints can be extracted along the activated path. Then all the data graphs are parsed as functional constraints at different states by using predicates. Finally, a DD model is represented as a single Prolog module [17].

3.4. Test Generation Algorithm

There are two types of tests which we consider in our current approach. One set targets nonterminal nodes of the control-flow DD (conditions for branch activation) and the second set aims at testing operators, depicted in terminal nodes of the data-flow DD. This approach enables us to test the behavior of the system and also explore information related to the final implementation of the system.

The whole test generation task is performed in the following way. Tests are generated sequentially for each nonterminal node of the control-flow DD. Symbolic path activation is performed and functional constraints are extracted. Solving the constraints gives us the path activation conditions to reach to the particular segment of the specification. In order to test the operations, presented in the terminal nodes of the data-flow DD, we employ a gate-level test pattern generator. In this way we can incorporate accurate structural information into the

high-level test pattern generation environment while keeping the propagation and justification task still on a high abstraction level. In the following chapter the test pattern generation algorithm is described in more detail.

3.4.1. Conformity Test

For the nonterminal nodes of the control-flow DD, conformity tests will be applied. The conformity tests target errors in branch activation. In order to test nonterminal node *IN1* in Figure 3, for example, one of the output branches of this node should be activated. Activation of the output branch means activation of a certain set of program statements. In our example, activation of the branch *IN1<0* will activate the branches in the data-flow DD where *q=1 (A:=X)*. For observability the values of the variables calculated in all the other branches of *IN1* have to be distinguished from the value of the variables calculated by the activated branch. In our example, node *IN1* is tested, in the case of *IN1<0*, if $X \neq Y$. The path from the root node of the control-flow DD to the node *IN1* has to be activated to ensure the execution of this particular specification segment and the conditions, generated here, should be justified to the primary inputs of the module. This process will be repeated for each output branch of the node. In the general case there will be *n(n-1)* tests, for every node, where *n* is the number of output branches.

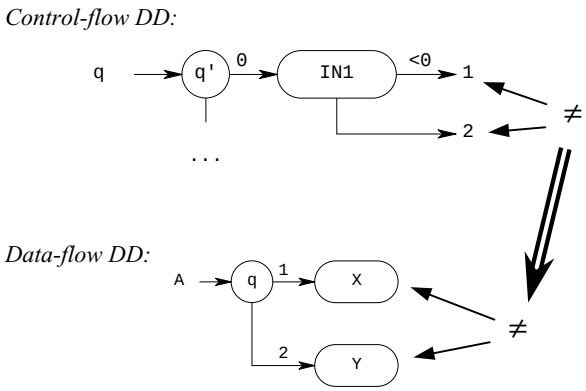


Figure 3: Conformity test

3.4.2. Testing Arithmetic Operators

Synthesis is the translation of a behavioral representation of a design into a structural one. One of the most important parameters guiding the synthesis process is the technology that will be used in the final implementation. By defining the technology, we can have among other information also the implementation of the functional units that will be used in the final design. Our hierarchical test generation algorithm employs this structural information for generating tests and estimating testability of the final implementation when using one or another library of functional units (FUs).

Tests are generated in cooperation with low-level test pattern generators as depicted in Figure 4. The arithmetic operator test generation is performed one by one for every FU given in the specification as depicted in Figure 5, where an example of generating low-level tests for an adder is given.

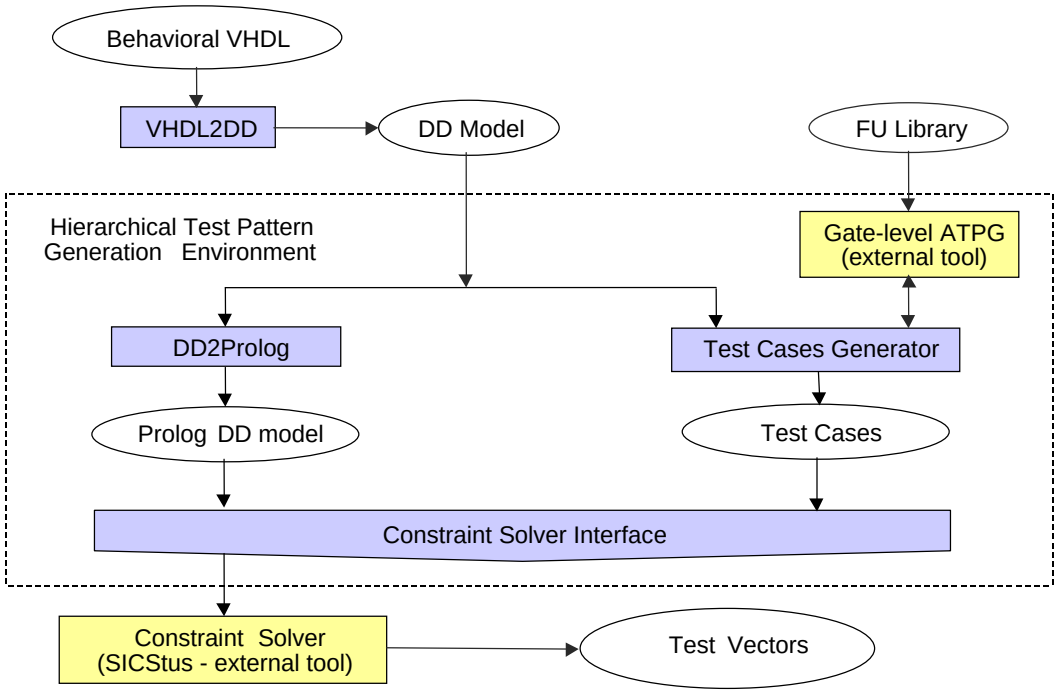


Figure 4: Hierarchical Test Generation Environment

We start by choosing a not tested operator from the specification and employ a gate level ATPG to generate a test pattern targeting structural faults in the corresponding FU. In our approach a PODEM like ATPG [6] is used but, in general, any arbitrary gate-level test pattern generation algorithm can be applied. If necessary, pseudorandom patterns can be used for this purpose as well. Test patterns which are generated by our current approach have typically some undefined bits. As justification and propagation are performed at the behavioral level by using symbolic methods those undefined bits have to be set to a defined value. Selecting the exact values is an important procedure as not all possible values can be propagated through the given behavior and can therefore have impact to the final fault coverage. The vectors that do not have any undefined bits are thereafter forwarded to the constraint solver, where together with the environmental constraints it is forming a test case. Solving such a test case guarantees that the generated low-level test vector can be justified till primary inputs and the fault effect is observable at primary outputs. If the constraint solver can not find an input combination, which would satisfy the given constraints, another combination of values for the undefined bits has to be chosen and the constraint solver should be employed again. This process is continued until a solution is found or timeout occurs. If there is no input combination which satisfies the generated test case, this particular low-level test pattern will be abandoned and gate-level ATPG will be employed again to generate a new low-level test pattern. This task is continued until low-level ATPG can not generate any more test patterns.

We are generating tests for every FU one by one and finally the fault coverage for every individual FU under given environmental constraints can be reported, which gives the possibility to rank all modules according to their testability. This ranking will be used in the next step, when modules for DfT modifications will be selected and self test structures will be inserted.

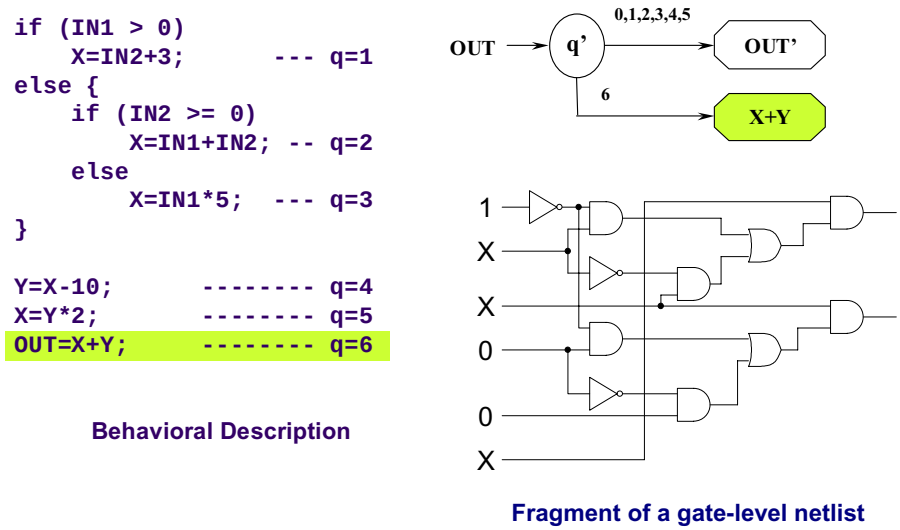


Figure 5: Testing Functional Units

4. High Level BIST Insertion

Built-in self-test is one of the mainstream DfT techniques and by appearance of SoC solutions it is making its way to the industrial deep submicron designs. The objective of our approach is to merge a Symbolic Testability Analysis (STA) based approach and novel HTG based approach to obtain an integrated framework for inserting self-test structures. In this framework, HTG will serve as a front-end tool for testability analysis and can provide preliminary information concerning hard to test modules. The framework will target a hybrid BIST architecture as the final implementation platform. By using the hybrid BIST architecture we can guarantee a solution which is cost effective and flexible.

Our current approach for high level BIST insertion is based on Symbolic Testability Analysis (STA) [14] extended with BIST based controllability and observability enhancement strategies and resource optimisation [13]. The STA based BIST approach provides solution to the problem of justification of test vectors and propagation of test responses by searching one or more control and observation paths, known as Test Environment (TE) for each FU. These TEs provide justification and propagation paths from on-chip test pattern generators and signature analysers to each separate FU.

The test environments for a FU under test, if such a TE exists, are obtained by looking at its input lines and tracing back the propagation paths that can be used to set its values from the primary input ports or pseudorandom pattern generators (PRPGs). To derive the test environments, it is necessary to force intermediate active functional modules to take particular values to assist in propagating test data from PRPGs to the FU under test and from the FU under test to an appropriate primary output or a multiple input signature register (MISR).

4.1. Problem Formulation

Initially, all primary input registers are converted into pseudorandom pattern generators (PRPG) and all primary output registers to multiple input signature registers (MISR). Additional BIST registers for self-test enhancement will then be used, if necessary. The particular problem we address is to optimize BIST resources usage under self-test time constraints. We aim at creating a tool to analyze the testability of the design and to determine the minimal possible testing time, T_{min}, which can be achieved as a result of the parallelism inherited by the nature of the design itself. Given a certain required maximum testing time, T_{req}, the following alternatives are taken:

- If $T_{req} < T_{min}$, return no solution;
- If $T_{req}=T_{min}$, optimize BIST hardware, so that minimal overhead is left and return the current testing time, T_{min}, and the modified RTL design;
- If $T_{req}>T_{min}$, optimize the BIST hardware, such that minimal overhead is left and testing time $T_{BIST} \leq T_{req}$, and return T_{BIST} and the modified RTL design.

The input to our BIST time analysis and resource optimization tool is an RTL design represented in Control Dataflow Graphs. The outputs are a test schedule, an RTL design with minimal added BIST resources and a merged design and BIST controller.

4.2. Proposed Methodology

STA defines four Boolean values for controllability and observability of each Test Control Data Flow (TCDF) variable. General controllability, $C_g(n)$, of a TCDF variable on the n^{th} control cycle is the ability to control the variable to any arbitrary value from the corresponding PRPGs. Similarly, controllability to the constant value 1, $C_1(n)$, and controllability to the constant value 0, $C_0(n)$, are defined. Observability, $O_v(n)$, of a TCDF variable V in n^{th} control cycle is the ability to observe any value of the variable at a MISR. If one or several of the controllability values needed to test a module, are *false*, then the associated variable is uncontrollable in the given control step.

Our main idea is illustrated in Figure 6. Inputs and outputs of the operations are variables, and the test environments of each operation are used to test the associated functional module that performs the operation. To test multiplier node *3 using PRPGs placed at the inputs of operations *1 and *2, and a MISR at the output of +4, we need to control V6 and V7 to general controllability values in the second control cycle and observe the value of V8 in the third control cycle. The test environments for operation *3 are given by *"$C_g(2)_{V6}$ and $C_g(2)_{V7}$ and $O_v(3)_{V8}$"* and are derived as follows. $C_g(2)_{V6}:=\{C_g(1)_{V1}$ AND $C_1(1)_{V2}\}$ OR $\{C_1(1)_{V1}$ AND $C_g(1)_{V2}\}$, $C_g(2)_{V7}:=\{C_g(1)_{V3}$ AND $C_1(1)_{V4}\}$ OR $\{C_1(1)_{V3}$ AND $C_g(1)_{V4}\}$ and $O_v(3)_{V8}:=O_v(4)_{V9}$ AND $C_0(1)_{V5}$. In total, there are four different alternative test environments for testing *3 (Table 1).

To illustrate our next idea, let us derive TEs for operation +5 (Figure 6) which are given as *"$C_g(2)_{V1}$ and $C_g(2)_{V4}$ and $O_v(3)_{V10}$"*. For left input, $C_g(2)_{V1}:=C_g(1)_{V1}$, for right input, $C_g(2)_{V4}:=C_g(1)_{V4}$ and for observability of output, $O_v(3)_{V10}:=O_v(4)_{V10}$.

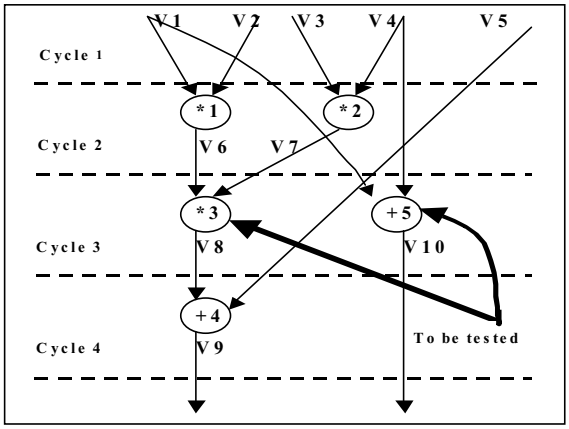


Figure 6: TCDF example.

	For controlling operation				*For observing responses*
TEs	**V1**	**V2**	**V3**	**V4**	**V5**
TE1 of op *3	$C_0(1)$	$C_1(1)$	$C_0(1)$	$C_1(1)$	$C_0(1)$
*TE2 of op *3*	*$C_g(1)$*	*$C_1(1)$*	*$C_1(1)$*	*$C_g(1)$*	*$C_0(1)$*
TE3 of op *3	$C_1(1)$	$C_g(1)$	$C_0(1)$	$C_1(1)$	$C_0(1)$
TE4 of op *3	$C_1(1)$	$C_0(1)$	$C_1(1)$	$C_0(1)$	$C_0(1)$
TE1 of op +5	$C_g(1)$	-	-	$C_g(1)$	-

Table 1: Alternative TEs for testing *3 and +5

If a given TCDF variable, say V_k, needs to be controlled to the same value in the same control cycle in test environments of different operations, say OP_{FU1}, OP_{FU2}… OP_{FUn}, then this common controllability value can be shared by those operations to perform their concurrent testing. For example, consider variables V1 and V4 in the TEs of *3 and +5. Both the second TE alternative of *3 (TE2 of op *3) and the TE of +5 need V1 and V4 to be controlled to $C_g(1)$. Therefore, V1 and V4 can be shared to perform concurrent testing of both operations using the test environment TE2 of operation *3. Our overall approach is described in Figure 7.

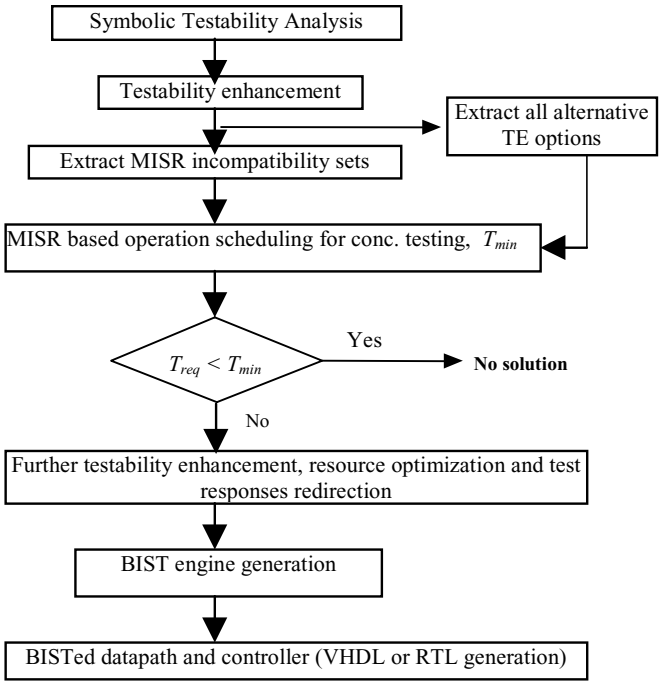


Figure 7: Overview of the BIST time analysis and resource optimization

4.3. BIST Synthesis

4.3.1. Initial Testability Enhancement

Uncontrollable nodes induce controllability problems to successor nodes. Our controllability enhancement strategy first enhances the node that is the source of controllability problems. Therefore, enhancing one node can improve controllability of most of the successor nodes. We do this by multiplexing the uncontrollable node with a node that is directly controllable from a primary input register or by adding a new dedicated PRPG and multiplexing it with the uncontrollable node.

Observability of an operation imposes restrictions on the values of other nodes (operations) in order for the test responses to be propagated to MISRs. If the restrictions are not able to force propagation of the values to MISR or some nodes are forced to have contradictory values simultaneously to enable observability then these operations become unobservable (Figure 8).

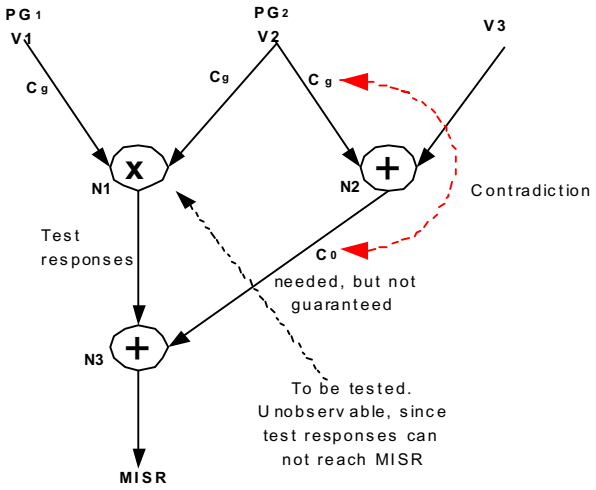


Figure 8: Observability problem due to contradictory values on intermediate nodes

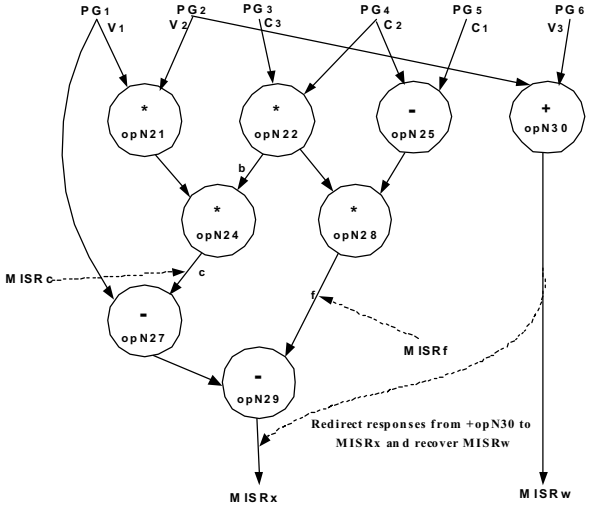


Figure 9: Enhancing observability of unobservable chains and MISR recovery strategy

If node N1 in Figure 8 is to be tested, controllability value C_g is to be set at V_1 and V_2 while the output of N2 has to be controlled to C_0 to enable observability of the output of N1 at a MISR. Since V_2 is also connected to N2, whatever value is set at V_3, C_0 can not be guaranteed at the output of N2, hence, test responses at the output of N1 can not reach the MISR.

One solution to the observability problem discussed above is to introduce a MISR at the output of the node N1 or redirect test responses from N1 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 strategy is to add a dedicated MISR at the output of a node situated at the end of a chain of unobservable nodes. If a MISR is added to an unobservable node that is not at the end of the unobservable chain,

then the downstream modules will still be unobservable. This idea is illustrated in Figure 9. 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 existence of two unobservable chains. The first one consists of nodes *opN21, *opN24 and *opN22 whereas the second consists of *opN22, *opN28 and –opN25. To enhance the observability of these chains, our approach selects to enhance observability of lines *c* and *f,* which are at the end of the first and second unobservable chains respectively. As a result, observability of all three nodes in each of the two chains is enhanced. Had we, for example, enhanced observability of lines *b* instead, only observability of node *opN22 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 is added into the design.

4.3.2. Alternative Test Environment Options

STA reveals the existence of possibly more than one TE for controlling input operands and observing test responses for each operation. If we want to observe node N1 in Figure 10, we need to observe arc A_{tbo} (A_{tbo} and A_{tbc} stand for arc to be observed and arc to be constrained to controllability value C_0 respectively). Based on STA, this implies constraining A_{tbc} to C_0, and observing the value of A_{tbo} at any of the observable output arc (A_{o1}, A_{o2},…, A_{on}) at the output of node N3. Therefore, the number of observability alternatives increases when the node N3 has multiple observability paths, which, in this case, are also inherited by the node N1, provided that A_{tbc} can be constrained to C_0.

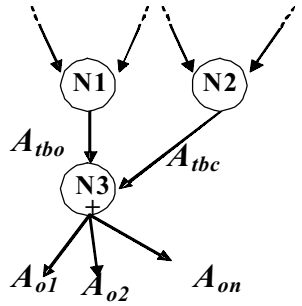


Figure 10: Multiple alternative observability paths

$_{alt}C_0(A_i)$ is defined as the number of compatible alternative test environment options (ATEO) that can be used to set arc A_i to a controllability value C_0. Similarly, we define $_{alt}C_1(A_i)$, and $_{alt}C_g(A_i)$. $_{alt}O(A_i)$ is the number of compatible alternative test environment options that can be used to enable observability of an arc A_i at some signature registers.

Two TE alternatives are compatible if and only if each of the TCDF variables 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. Therefore, the total number of observability alternative options for arc A_{tbo} can be derived as follows,

$$_{alt}O(A_{tbo}) = {}_{alt}C_0(A_{tbc}) \times {}_{alt}O(A_{o1}) + {}_{alt}C_0(A_{tbc}) \times {}_{alt}O(A_{o2}) + \ldots + {}_{alt}C_0(A_{tbc}) \times {}_{alt}O(A_{on}) \ldots (1)$$

$$= {}_{alt}\ C_0(A_{tbc}) \times \sum_{i=1}^{n} {}_{alt}\ O(A_{oi})$$

Out of these alternatives, that particular TE alternative option which minimizes MISR conflicts and can lead to packing as many operations as possible in each test session will be chosen during test session selection process. Consequently, the total number of test sessions will be minimized. In addition, TEs of all operations in a test session must be simultaneously

supported. When the best choice of TE alternatives that give the smallest feasible number of test sessions is achieved, the associated testing time is known as minimal testing time, T_{min}.

4.3.3. MISR Incompatibility Sets (MISRISs)

MISR incompatibility sets consist of operations that cannot be tested concurrently 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 tested concurrently.

To extract MISRISs we group operations based on the MISRs that are used to analyze their responses. Each MISR, M_i, corresponds to one set, G_i, which will include all operations that it analyzes. All operations in the same set are known as MISR incompatible operations.

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 TE options, which determine whether the TEs are conflict free.

When an operation has multiple independent observability paths it will be included in more than one MISRIS. As discussed in previous section, the operation can also have many alternative test environments. We have investigated how different choices of the alternative observability paths impact TE complexity and the resulting number of test sessions. Based on the results we can develop a heuristic to choose the best observability path if a FU is observable through multiple observability paths.

4.3.4. Concurrent Test Session Selection

Concurrent test sessions are selected based on MISRIS. A group consisting of one operation from each MISRIS can possibly be tested concurrently if their TEs constraints are not in conflict.

If the TE constraints are not considered, it can be possible to schedule 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 at the same time for all operations in a given test session. In this way, controllability constraints imposed by the TEs 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.

TEs have two components. The first component consists of the controllability values necessary to justify the inputs of the operations and the second component consists of the controllability values necessary to propagate test responses to the corresponding MISR. Thus, when constraints required to justify input operands and those imposed to propagate test responses to appropriate MISR are taken into account during test session selection process, an increase in the number of test sessions can be noticed. Consequently, 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

minimal testing time, T_{min}. Thus, it is possible to test the design in T_{min} test sessions as a result of the nature of parallelism inherited from the design itself.

Our heuristic for selecting concurrent test sessions is based on an equal length test-scheduling algorithm [2]. 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 test session not only if they do not share MISR, but also if controllability and observability constraints are simultaneously satisfied for all of them.

4.4. BIST Resources Optimization

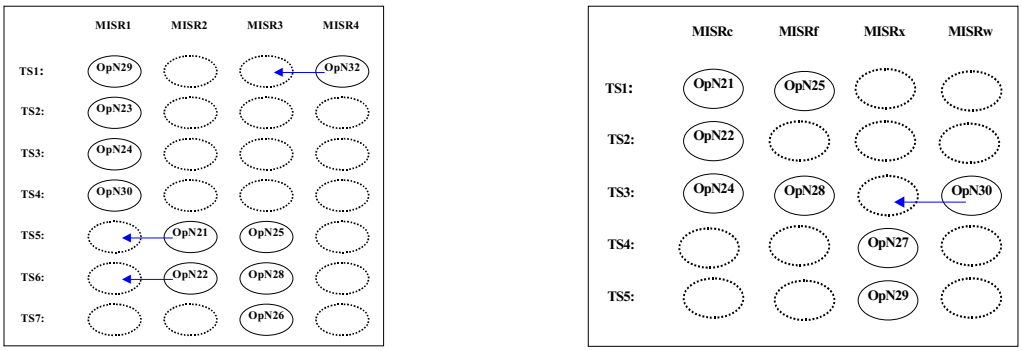
MISRs are not effectively used in some test sessions. Our approach recovers some MISRs and converts them back to normal registers. The operations that use recovered MISRs are redirected to other free MISRs in the same test session.

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 called X, G_X represents the incompatibility set corresponding to MISR X.

The algorithm below minimizes the set M_C of used MISRs and produces the corresponding incompatibility sets. This optimization is performed without increasing the number of test sessions.

```
Begin
    G   set of all incompatibility sets; Best_selection_obtained   FALSE;
    While (best_selection_obtained != TRUE)
      Lu   X, X ∈ Mc and X is least used; U   All test sessions in which Lu is used;
      F   Free MISRs in sessions U;
      If F ≠ϕ  then
          P   X, X ∈ F and X is most packed;
          G   G – { GP, GLu };  GP   GP U GLu;  G   G U { GP }; MC   MC - { Lu };
      else
          best_selection_obtained   TRUE;
    return MC, G;
End.
```

The figures below illustrate how the MISR recovery algorithm works. For Diff example MISR4 is recovered and test responses for opN32 are redirected to MISR3. Similarly, for the EX design, MISRw is recovered and test responses from opN30 are redirected to MISRx.



MISR recovery for Diff. Design MISR recovery for EX design

Figure 11: MISR recovery algorithm

4.5. Combined Use of HTG and STA for BIST Insertion

Our approach for testability enhancement and BIST insertion is based on combined use of HTG and STA as discussed above. We can use the results from HTG to group FUs into three sets.

The first set consists of FUs which testability is above the user-defined threshold fault coverage. These FUs will be tested together in a single test session with the long LFSR situated at the primary input and a long MISR at the primary output. FUs in this set can be tested in the first test session so that the test vectors will also be indirectly used to capture some of the faults in the next sets of FUs as discussed below.

The second set consists of FUs which testability is below threshold fault coverage. The FUs in this set will be tested based on our STA based BIST testing methodology. Alternative TEs for each of the FU in this group will be searched and appropriate test sessions selected and finally BIST resources for these FUs will be optimized. We also look at the possibilities of reducing the length of the test sessions in this group by taking into account some of the test vectors that were applied when testing the FUs from the first set. In other words, faults that are already indirectly detected in the first session will not be considered, hence, only the remaining faults will be targeted. Appropriate seed and polynomials for minimizing the number of test vectors for detecting the few remaining faults can also be appropriately selected.

The third set consists of random resistant FUs. These FUs have to be tested by using deterministic test vectors. FUs in this set will also be tested by using STA based approach to guarantee that deterministic test vectors are supplied to appropriate FUs.

Finally, a global total testability (GTT) metric to compute total FC of the whole design can be used. The value of GTT can fine-tune the results by making a good trade-off between TE complexity and BIST overhead when using HTG and STA approaches.

5. Hybrid BIST

A typical BIST architecture consists of a test pattern generator (TPG), a test response analyzer (TRA) and a BIST control unit (BCU), all implemented on the chip. The classical way to implement the TPG for logic BIST (LBIST) is to use linear feedback shift registers

(LFSR). But as the test patterns generated by the LFSR are pseudorandom by their nature, the LFSR-based approach often does not guarantee a sufficiently high fault coverage (especially in the case of large and complex designs) and demands very long test application times in addition to high area overheads. Therefore, several proposals have been made to combine pseudorandom test patterns, generated by LFSRs, with deterministic patterns, to form a hybrid BIST solution.

In our approach we propose to use a hybrid test set, which consists of a limited number of pseudorandom and deterministic test vectors. The main idea is to first apply a limited number of pseudorandom test vectors, which is followed by the application of the stored deterministic test set specially designed to shorten the pseudorandom test cycle and to target the random resistant faults. The basic idea of Hybrid BIST is discussed in [7].

In our STA based BIST approach we proposed a methodology for analyzing testability of the design and proposed an approach to enhance testability to get 100% controllability and observability for each of the functional modules. Initially it was done by converting primary input registers to PRPGs and primary output registers to MISRs. Further testability enhancement demanded conversion of some of the inner registers to BIST registers. Though some of the MISR registers could be recovered after our resource optimization approach was used, still we had to pay for BIST area and wiring overhead.

5.1. Hybrid BIST architecture

To reduce the hardware overhead in the LBIST architectures the hardware LFSR implementation can be replaced by software, which is especially attractive to test SoCs, because of the availability of computing resources directly in the system (a typical SoC usually contains at least one processor core). On the other hand, the software based approach implies large memory requirements (to store the test program and test patterns).

A hardware based hybrid BIST architecture is depicted in Figure 12, where the pseudorandom pattern generator (PRPG) and the Multiple Input Signature Analyzer (MISR) are implemented inside the core under test (CUT). The deterministic test pattern are precomputed off-line and stored inside the system.

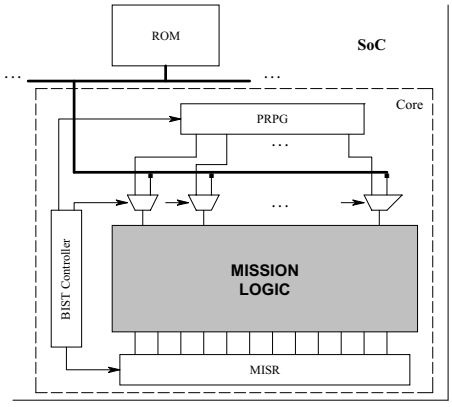


Figure 12: Hardware based hybrid BIST architecture

To avoid the hardware overhead caused by the PRPG and MISR, and the performance degradation due to excessively large LFSRs, a software based hybrid BIST can be used where pseudorandom test patterns are produced by the test software. Our the cost calculation and optimization algorithms are general, and can be applied as well to the hardware based as to the software based hybrid BIST optimization.

In case of the software based solution, the test program, together with test data (LFSR polynomials, initial states, pseudorandom test length, signatures), is kept in a ROM. The deterministic test vectors are generated during the development process and are stored in the

same place. For transporting the test patterns, we assume that some form of test access mechanism is available.

In test mode the test program is executed in the processor core. The test program proceeds in two successive stages. In the first stage the pseudorandom test pattern generator, which emulates the LFSR, is executed. In the second stage the test program will apply precomputed deterministic test vectors to the core under test.

The pseudorandom TPG software is the same for all cores in the system and is stored as one single copy. All characteristics of the LFSR needed for emulation are specific to each core and are stored in the ROM. They will be loaded upon request. Such an approach is very effective in the case of multiple cores, because for each additional core, only the BIST characteristics for this core have to be stored. The general concept of the software based pseudorandom TPG is depicted in Figure 13.

Although it is assumed that the best possible pseudorandom sequence is used, not always all parts of the system are testable by a pure pseudorandom test sequence. It can also take a very long test application time to reach a good fault coverage level. In case of hybrid BIST, we can dramatically reduce the length of the initial pseudorandom sequence by complementing it with deterministic stored test patterns, and achieve the 100% fault coverage.

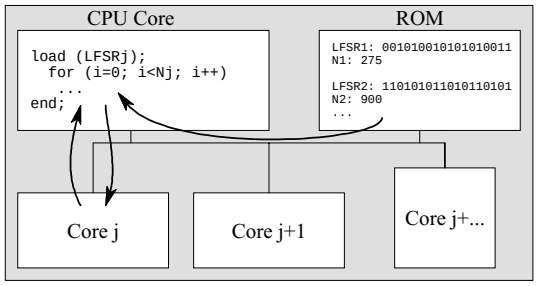


Figure 13: LFSR emulation

Our previous approach required insertion of additional hardware based PRPGs and MISRs. When using a hybrid BIST approach those modifications are not required and only additional wiring for transporting pseudo-random and deterministic test vectors to inner locations of the circuitry for testability enhancement is needed. Similarly, test responses can be directed from internal unobservable lines to the software based signature analyzer. Thus, in our new testability and enhancement framework, we use our BIST based STA approach to pinpoint hard to test functional modules in the design and instead of adding additional test structures to the places suggested by our testability analysis, we use wiring transformation to enhance controllability and observability. This gives us significant reduction of hardware overhead while providing a mechanism for testing random resistant faults and consequently increasing test quality.

5.2. Cost calculation for Hybrid BIST

In a hybrid BIST technique the length of the pseudorandom test L is an important parameter, which determines the behavior of the whole test process. It is assumed in this report that for the hybrid BIST the best polynomial for the pseudorandom sequence generation will be chosen. Removing the latter part of the pseudorandom sequence leads to a lower fault coverage achievable by the pseudorandom test. The loss in fault coverage should be covered by additional deterministic test patterns. In other words, a shorter pseudorandom test set implies a larger deterministic test set. This requires additional memory space, but at the same time, it shortens the overall test process. A longer pseudorandom test, on the other hand, will lead to longer test application time with reduced memory requirements. Therefore

it is crucial to determine the optimal length of the pseudorandom test L_{OPT} in order to minimize the total testing cost.

Figure 14 illustrates the total cost calculation for the hybrid BIST consisting of pseudorandom test and stored test, generated off-line. We can define the total test cost of the hybrid BIST C_{TOTAL} as:

$$C_{TOTAL} = C_{GEN} + C_{MEM} = \alpha L + \beta S \quad (2)$$

where C_{GEN} is the cost related to the time for generating L pseudorandom test patterns (number of clock cycles), C_{MEM} is related to the memory cost for storing S precomputed test patterns to improve the pseudorandom test set, and α, β are constants to map the test length and memory space to the costs of the two parts of the test solutions to be mixed. Figure 14 illustrates how the cost of pseudorandom test is increasing when striving to higher fault coverage (the C_{GEN} curve). The total cost C_{TOTAL} is the sum of the above two costs. The weights α and β reflect the correlation between the cost and the pseudorandom test time (number of clock cycles used) and between the cost and the memory size needed for storing the precomputed test sequence, respectively.

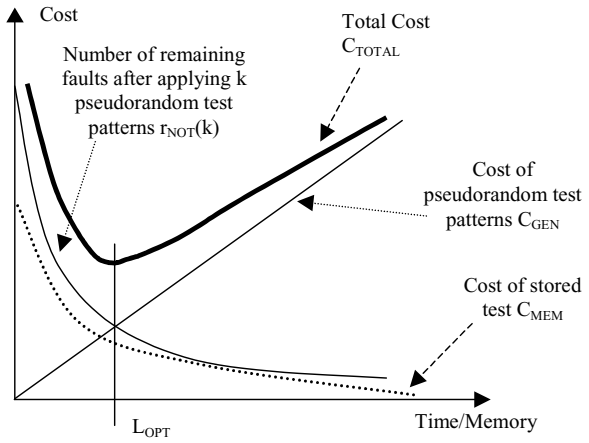


Figure 14: Cost calculation for hybrid BIST

Creating the curve $C_{GEN} = \alpha L$ is not difficult. For this purpose, the cumulative fault coverage for the pseudorandom sequence generated by a LSFR should be calculated by a fault simulation. More difficult is to find the values for $C_{MEM} = \beta S$. For this purpose we can use either ATPG based or fault table based approach and for reducing the number of calculations a Tabu search based optimization algorithm is proposed [8].

6. Experimental Results

The main goal of the COTEST project was assessment of feasibility and effectiveness of high-level DfT modifications. For this purpose we have developed a prototype environment for evaluating several key techniques, which serve as a platform for future developments. The environment consists of our in-house tools, external academic as well as commercial tools and is depicted in Figure 15. For high-level synthesis the in-house high-level synthesis tool CAMAD [4] is utilized. Subsequent logic synthesis is performed by AutoLogic II from Mentor Graphics [11] and stuck-at fault coverage at gate level is measured with the Turbo

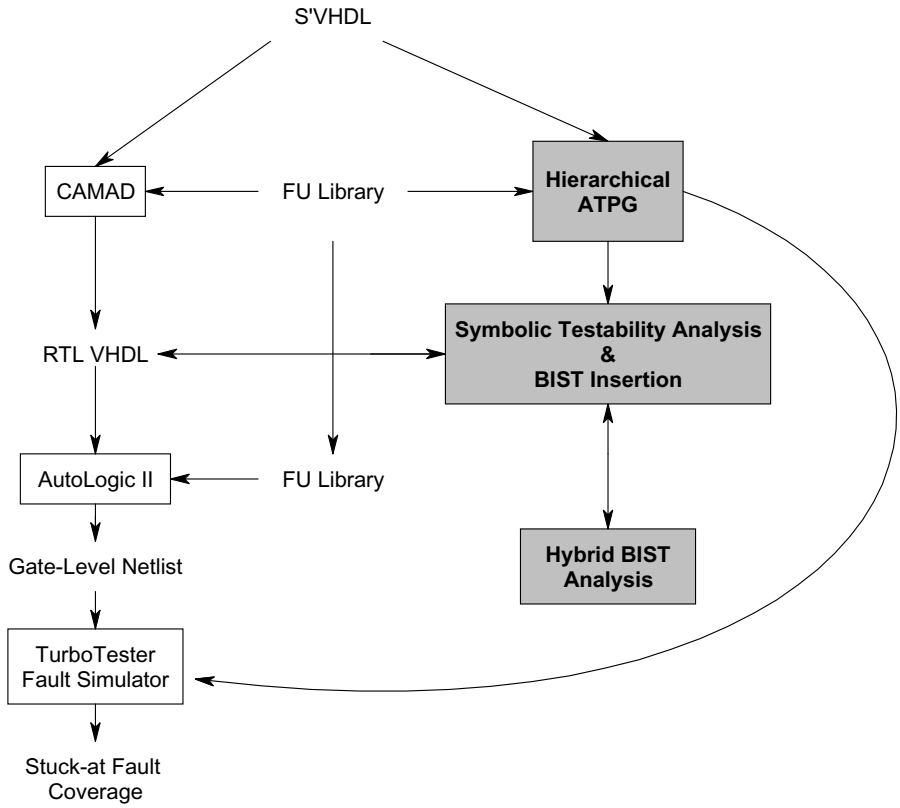


Figure 15: Experimental environment

Tester toolset from Tallinn Technical University [6]. Additionally the library of functional units was developed and is available at behavioral and gate levels as well as a set of compilers for different intermediate formats.

6.1. Hierarchical ATPG

We have investigated possibilities to incorporate structural information into the behavioral level test generation environment. Some of those results are already discussed in the COTEST report D2 [5]. In the following we are presenting and discussing results obtained with benchmark circuits defined in the COTEST report D1 [16].

We have investigated possibilities to apply our ATPG approach for industrial control-dominated design F4 [16]. We have extracted two blocks from the F4 specification: *F4_InputHandler_1* and *F4_OutputHandler_1*. Results are compared with commercial gate-level ATPG tool from Mentor Graphics (FlexTest [12]) and presented in Table 2:

Design	VHDL Lines [#]	Stuck-at faults [#]	**High-level HTG**			Gate-level ATPG		
			Test length	**CPU [s]**	**FC [%]**	Test length	CPU [s]	FC [%]
F4_InputHandler_1	175	4872	**62**	**228**	**64.22%**	219	811	38.22%
F4_OutputHandler_1	54	872	**26**	**1.52**	**76.26%**	170	5	81.30%

Table 2: ATPG results with F4 design

As it can be seen, HTG can produce results which are comparable with results obtained at the gate-level, while having shorter test generation time and reduced test length.

Another set of experiments was performed with the DIFFEQ design [16], which belongs to the HLS'92 benchmark suite. This design was analyzed more thoroughly for obtaining data for BIST insertion and the results are presented in Figure 16 and summarized in Table 3.

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

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

    while x_var < a_var loop

       t1 := u_var * dx_var;
       -- Tested 5634 faults
       -- Untestable 0
       -- Aborted 14
       -- Fault coverage: 99.752125
       -- Fault efficiency: 99.752125
       -- 52 Vectors

       t2 := x_var * 3;
       -- Tested 4911 faults
       -- Untestable 0
       -- Aborted 737
       -- Fault coverage: 86.951133
       -- Fault efficiency: 86.951133
       -- 11 Vectors

       t3 := y_var * 3;
       -- Tested 4780 faults
       -- Untestable 0
       -- Aborted 868
       -- Fault coverage: 84.631728
       -- Fault efficiency: 84.631728
       -- 10 Vectors

       t4 := t1 * t2;
       -- Tested 5621 faults
       -- Untestable 0
       -- Aborted 27
       -- Fault coverage: 99.521955
       -- Fault efficiency: 99.521955
       -- 38 Vectors

       t5 := dx_var * t3;
       -- Tested 5616 faults
       -- Untestable 0
       -- Aborted 32
       -- Fault coverage: 99.433428
       -- Fault efficiency: 99.433428
       -- 35 Vectors

       t6 := u_var - t4;
       -- Tested 368 faults
       -- Untestable 0
       -- Aborted 60
       -- Fault coverage: 85.981308
       -- Fault efficiency: 85.981308
       -- 9 Vectors

       u_var := t6 - t5;
       -- Tested 424 faults
       -- Untestable 0
       -- Aborted 4
       -- Fault coverage: 99.065421
       -- Fault efficiency: 99.065421
       -- 15 Vectors

       t7 := u_var * dx_var;
       -- Tested 1123 faults
       -- Untestable 0
       -- Aborted 4525
       -- Fault coverage: 19.883144
       -- Fault efficiency: 19.883144
       -- 1 Vectors

       y_var := y_var + t7;
       -- Tested 389 faults
       -- Untestable 0
       -- Aborted 39
       -- Fault coverage: 90.887850
       -- Fault efficiency: 90.887850
       -- 11 Vectors

       x_var := x_var + dx_var;
       -- Tested 414 faults
       -- Untestable 0
       -- Aborted 14
       -- Fault coverage: 96.728972
       -- Fault efficiency: 96.728972
       -- 15 Vectors

     end loop ;

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

     END PROCESS ;

END behavior;
```

Figure 16: DIFFEQ benchmark with testability figures for every individual functional unit

In Figure 16 we have associated with every FU a total number of stuck-at faults in the FU, when implemented in the target technology, a number of vectors, which were generated by a gate-level ATPG and successfully justified till primary inputs and propagated till primary outputs and the final stuck-at fault coverage for every FU. As it can be seen, fault coverage of functional units differs significantly, depending of the location and type of every individual FU. This information will be exploited at the latter stage of DfT flow, when selecting modules for BIST insertion. In Table 3 the comparative results for the whole DIFFEQ design are presented. Those results are discussed in detail in the COTEST report D2 [5].

	High level ATPG			**Hierarchical ATPG**			`testgen`		
	FC [%]	Len [#]	CPU [s]	FC [%]	Len[1] [#]	CPU [s]	FC [%]	Len [#]	CPU [s]
DIFFEQ 1	97.25	553	954	98.05	199	468	99.62	1,177	4,792
DIFFEQ 2	94.57	553	954	96.46	199	468	96,75	923	4,475

Table 3: Summarized HTG results for DIFFEQ benchmark circuit.

6.2. High-level BIST insertion

For high-level BIST insertion we performed experiments with several designs, including COTEST benchmark design DIFFEQ. We have chosen more designs than included in the COTEST benchmark report [16] for obtaining additional data for comparisons. At first the standard STA based approach was applied and thereafter testability was enhanced till 100%. The results are depicted in Table 4, where the first column gives information about the design (name as well as number and type of functional units), the second column shows the number

Design		Test sessions (T_{min})	Applying original STA				**100% Testability enhancement approach**			
					%Testability		**Straightforward**		**Optimized**	
Name	Operations		#PG	#MISR	Contr.	Observ.	#PGs	#MISRs	#PGs	#MISRs
Diffeq	**2+, 2-, 6***	**7**	**4**	**3**	**40**	**70**	**5**	**4**	**5**	**2**
Ex	1+, 3-, 4*	5	3	2	12.5	37.5	6	4	6	3
Tseng	4+, 2*, 1 /, 1 &	4	5	3	100	37.5	5	5	5	3
Paulin	2+, 2-, 6*	6	4	3	50	50	5	5	5	3

Table 4: BIST resources after applying STA and testability enhancement and optimization to 100% testability

[1] The number of test patterns simulated at the gate-level is larger due to the fact that every behavioral-level test pattern has to be expanded on gate-level over multiple timeframes.

of test sessions, and the third column shows the number of PGs, MISRs and the testability of the design after applying STA, but before testability enhancement. Controllability and observability sub-columns depict testability. Testability is computed as a percentage of controllable or observable operations. Controllability is the ratio of the number of controllable operations to the total number of operations in the design. Observability is the ratio of the number of observable operations to the total number of operations. An operation is controllable if both its left and right hand operands are simultaneously controllable. If any input operand is not controllable the associated operation is not controllable. At the latter part of the table the number of PGs and MISRs, after testability is enhanced to 100%, is given. At first the number of PGs and MISRs after a straightforward testability enhancement is presented and thereafter the number of PGs and MISRs, that remain in the design after applying our BIST resource optimization and MISR recovery strategy is given. The results show that by careful BIST optimization at the high level, the needed area overhead can be reduced (between 25% and 50% in terms of MISRs).

6.3. Hybrid BIST

For the hybrid BIST approach experiments were carried out on the ISCAS'85 benchmark circuits for comparing different algorithms, and for investigating the efficiency of the Tabu search method for optimizing the hybrid BIST. ISCAS'85 circuits were chosen for comparative purposes, as those circuits are thoroughly investigated and experimental data is available in the literature. Experiments were carried out using the Turbo Tester toolset [15] for deterministic test pattern generation, fault simulation, and test set compaction. The results are presented in Table 5 and illustrated by a diagram in Figure 17.

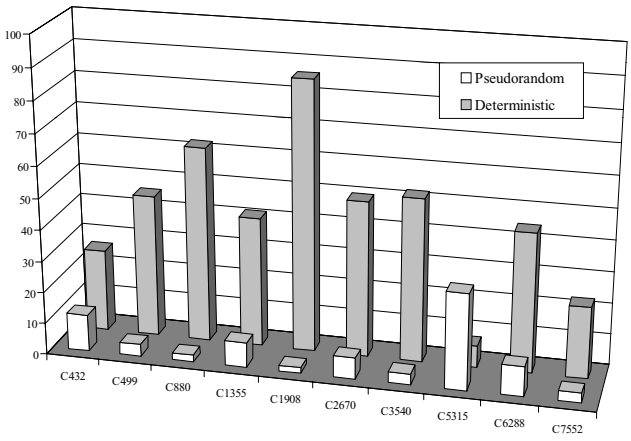


Figure 17: Percentage of test patterns in the optimized test sets compared to the original test sets

In the columns of Table 5 the following data is depicted: ISCAS'85 benchmark circuit name, *L* - length of the pseudorandom test sequence, F*C* - fault coverage, *S* - number of test patterns generated by deterministic ATPG to be stored in BIST, *Cost* – total cost of BIST

In Figure 17 the amount of pseudorandom and deterministic test patterns in the optimal BIST solution is compared to the sizes of pseudorandom and deterministic test sets when only either of the sets is used. In the typical cases less than half of the deterministic vectors and

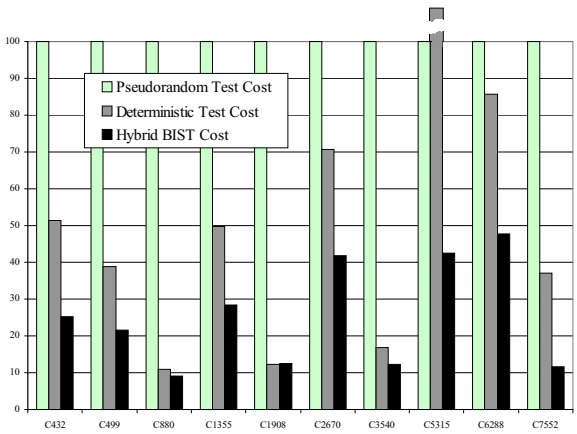


Figure 18: Cost comparison of different methods. Cost of pseudorandom test is taken as 100%

Circuit	*Pseudorandom test*		*Stored test*		*Hybrid test*		
	L	*FC*	*S*	*FC*	*L*	*S*	*Cost*
C432	780	93.0	80	93.0	91	21	196
C499	2036	99.3	132	99.3	78	60	438
C880	5589	100.0	77	100.0	121	48	505
C1355	1522	99.5	126	99.5	121	52	433
C1908	5803	99.5	143	99.5	105	123	720
C2670	6581	84.9	155	99.5	444	77	2754
C3540	8734	95.5	211	95.5	297	110	1067
C5315	2318	98.9	171	98.9	711	12	987
C6288	210	99.3	45	99.3	20	20	100
C7552	18704	93.7	267	97.1	583	61	2169

Table 5: Hybrid BIST Experimental results

only a small fraction of pseudorandom vectors are needed, however the maximum achievable fault coverage is guaranteed and achieved. Figure 18 compares the costs of different approaches using for Hybrid BIST cost calculation an equation 2 with the parameters $\alpha = 1$, and β = B where B is the number of bytes of the input test vector to be applied on the CUT. As pseudorandom test is usually the most expensive method. It has been selected as a reference and given value 100%. The other methods give considerable reduction in terms of cost and, as it can be seen, the hybrid BIST approach has significant advantages compared to the pure pseudorandom or stored test approach, in most of the cases.

7. Conclusions

The main goal of this assessment project was the evaluation of different test and DfT methodologies at higher levels of abstraction. With our work we have experimentally demonstrated that it is feasible to reason about testability and to introduce DfT structures at early phases of the design cycle. Particularly we obtained high quality results by incorporating some structural information during the analysis at the behavioral level. We have investigated the possibilities of inserting self-test structures, as one of the dominating low-level DfT methodologies, in the early phases of the design flow. By introducing a hybrid BIST architecture we have increased the flexibility of our approach leading to very attractive SoC solutions.

As it was demonstrated with our HTG algorithm, working at high levels of abstraction allows to reduce the test generation effort by a factor ranging from 3 to 10, while keeping the same high quality. By inserting self-test structures our approach guarantees full (100%) testability, while reducing area overhead in terms of signature analyzers compare to the straightforward solution by 25% to 50%. This area overhead can further be reduced by using a hybrid BIST architecture, where some of the required test structures can be implemented in software. As it was demonstrated experimentally, with a hybrid BIST solution less than 50% of deterministic patterns and only a small fraction of pseudorandom vectors are needed, while the maximum achievable fault coverage is guaranteed.

8. References

[1] A. K. Chandra, V. S. Iyengar, ”Constraint Solving for Test Case Generation: A Technique for High-level Design Verification”, *Computer Design: VLSI in Computers and Processors*, ICCD '92, pp.245 –248, 1992.

[2] G. L. Craig, C. R. Kime and K. K. Saluja, “Test Scheduling and Control for VLSI Built-In Self-Test”, *IEEE Transactions on Computers*, 1988.

[3] P. Eles, K. Kuchcinski, Z. Peng, M. Minea, “Compiling VHDL into a High-Level Synthesis Design Representation”, Proc. EURO-DAC with EURO-VHDL, 1992

[4] P. Eles, K. Kuchcinski, Z. Peng, “System Synthesis with VHDL”, Kluwer Academic Publishers, Boston, Dec. 1997, 384 pages.

[5] O. Golubaeva, M. Sonza Reorda, M. Violante, “COTEST Report D2: Report on automatic generation of test benches from system-level descriptions”, Politecnico di Torino, 2002.

[6] G.Jervan, A.Markus, P.Paomets, J.Raik, R.Ubar, “A CAD System for Teaching Digital Test”, *2nd European Workshop on Microelectronics Education*, Kluwer Academic Publishers, pp. 287-290, the Netherlands, 1998.

[7] G. Jervan, Z. Peng, R. Ubar, “Test Cost Minimization for Hybrid BIST”, *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'2000)*, pp. 283-291, Japan, 2000.

[8] G. Jervan, Z. Peng, R. Ubar, H. Kruus, “A Hybrid BIST Architecture and its Optimization for SoC Testing”, *IEEE 2002 3rd International Symposium on Quality Electronic Design (ISQED'02)*, pp. 273-279, USA, 2002.

[9] K. Kuchcinski, ”An Approach to High-level Synthesis using Constraint Logic Programming”, *24th Euromicro Conference*, Vol. 1, pp.74-82, Sweden, 1998.

[10] K. Marriott, P.J. Stuckey, Programming with Constraints: Introduction, *MIT Press*, 1998.

[11] Mentor Graphics, AutoLogic VHDL Synthesis Guide, *Mentor Graphics*, 1993.

[12] Mentor Graphics, FlexTest User's and Reference Manual, *Mentor Graphics*, 1998.

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

[14] S. Ravi, G. Lakshminarayana, N. K. Jha, “TAO: Regular Expression based High-Level Testability Analysis and Optimization”, International Test Conference (ITC’98), pp. 331 –340, USA, 1998.

[15] SICStus Prolog User's Manual, Swedish Institute of Computer Science, 2001.

[16] M. Sonza Reorda, M. Violante G. Jervan, Z. Peng, “COTEST Report D1: Report on benchmark identification and planning of experiments to be performed”, Politecnico di Torino, 2002.

[17] Y. Sun, “Automatic Behavioral Test Generation By Using a Constraint Solver”, *Final Thesis*, LiTH-IDA-Ex-02/13, Linköping University, 2001.

[18] R. Ubar, “Test Generation for Digital Circuits Using Alternative Graphs”, *Proc. of Tallinn Technical University*, Estonia, No. 409, pp. 75-81 (in Russian), 1976.

[19] R. Ubar, “Test Synthesis with Alternative Graphs”, *IEEE Design and Test of Computers*, Vol. 13, No. 1, pp. 48-57, Spring 1996.

