The Dissertation Committee for Mahesh Prabhu
certifies that this is the approved version of the following dissertation:

Scalable Algorithms for Software Based Self Test using Formal Methods

Committee:

Jacob A. Abraham, Supervisor

Jayanta Bhadra

Andreas Gerstlauer

David Z. Pan

Nur A. Touba

Scalable Algorithms for Software Based Self Test using Formal Methods

by

Mahesh Prabhu, B.E., M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2014

Dedicated to my parents and my wife.

Acknowledgments

I would like to thank my advisor Dr. Jacob Abraham for his valuable guidance and support. His novel ideas, intellectual discussions and expert comments at every stage have been the catalysts for this work.

I thank all my PhD committee members for giving me valuable inputs on my research and on this dissertation.

Sincere thanks to the students at CERC who were generously available to answer my questions and to direct me to right resources whenever needed. I would especially like to thank Gaurav, Sriram, Ameya, Shahrzad, EJ, Whitney and Junyoung.

Thanks to the CERC and ECE department staff members for their quick response in all the matters that needed their support. I thank Melanie, Debi and Andrew for answering all my questions patiently.

I thank my brother and my sister-in-law for their love and support throughout my PhD. Thanks to my parents for their selfless love and faith in me. Without their support and understanding this PhD would not have been possible. Last but not the least, I thank my loving wife, Shwetha, for standing by me through the most difficult of times.

Scalable Algorithms for Software Based Self Test using Formal Methods

Mahesh Prabhu, Ph.D.
The University of Texas at Austin, 2014

Supervisor: Jacob A. Abraham

Transistor scaling has kept up with Moore's law with a doubling of the number of transistors on a chip. More logic on a chip means more opportunities for manufacturing defects to slip in. This, in turn, has made processor testing after manufacturing a significant challenge. At-speed functional testing, being completely non-intrusive, has been seen as the ideal way of testing chips. However for processor testing, generating instruction level tests for covering all faults is a challenge given the issue of scalability. Data-path faults are relatively easier to control and observe compared to control-path faults. In this research we present a novel method to generate instruction level tests for hard to detect control-path faults in a processor. We initially map the gate level stuck-at fault to the Register Transfer Level (RTL) and build an equivalent faulty RTL model. The fault activation and propagation constraints are captured using Control and Data Flow Graphs of the RTL as a Liner Temporal Logic (LTL) property. This LTL property is then negated and given to a

Bounded Model Checker based on a Bit-Vector Satisfiability Module Theories (SMT) solver. From the counter-example to the property we can extract a sequence of instructions that activates the gate level fault and propagates the fault effect to one of the observable points in the design. Other than the user supplying instruction constraints, this approach is completely automatic and does not require any manual intervention.

Not all the design behaviors are required to generate a test for a fault. We use this insight to scale our previous methodology further. Under-approximations are design abstractions that only capture a subset of the original design behaviors. The use of RTL for test generation affords us two types of under-approximations: bit-width reduction and operator approximation. These are abstractions that perform reductions based on semantics of the RTL design. We also explore structural reductions of the RTL, called path based search, where we search through error propagation paths incrementally. This approach increases the size of the test generation problem step by step. In this way the SMT solver searches through the state space piecewise rather than doing the entire search at once. Experimental results show that our methods are robust and scalable for generating functional tests for hard to detect faults.

Table of Contents

List of Tables

List of Figures

Chapter 1

Testing: An introduction

Advances in VLSI fabrication technology has maintained the trend of doubling the number of transistors on a chip every 18 months (Moore's law [51]). This is mainly due to the decrease in the sizes of transistors and wires from the micron scale to the nanometer scale. Smaller transistors enables designers to cram in more logic into a single chip, while at the same time affording increased operating frequencies. However, the downside to having more logic on a chip is the increased probability that the chip might have a defect. Avoiding defects in the manufacturing process is an impossible task and some percentage of the chips will be faulty (resulting in *yield loss*). VLSI testing targets this important stage, in the process of going from an abstract design to silicon, to where we need to identify faulty chips. Furthermore, VLSI testing might also involve identifying the cause of the defect since it will help to improve the yield at various stages of chip manufacture [71].

In the VLSI development process, a design is transformed from the highest level of abstraction into silicon through a series of steps. Bugs or errors can be introduced during the transformation due to various reasons. Hence, at each step we verify if design matches the abstraction used in the previous step.

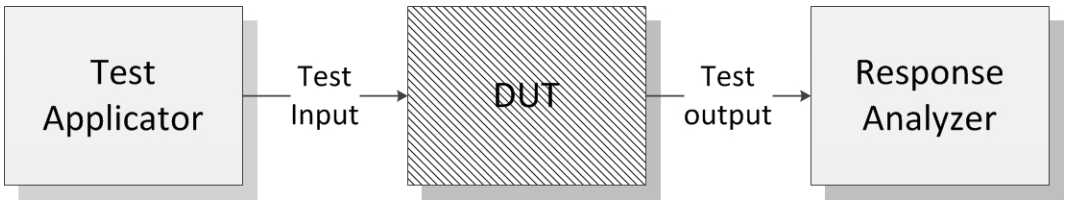

Figure 1.1: Testing infrastructure

Verification ensures that the design in its current form of abstraction meets the specification. The design can be the gate level netlist and the specification can be the register transfer level (RTL) design. Testing is similar to verification, in both the steps we compare two different abstractions of the design. In testing, the fabricated silicon is "verified" with the gate level netlist as the specification. The focus of this research is VLSI manufacturing test.

Manufacturing test typically involves the application of stimuli at the inputs of the design under test (DUT) and the analysis of the corresponding response at the output as shown in Figure 1.1. The response at the output is compared against the golden output from a fault free circuit and an unexpected mismatch would suggest a faulty DUT. The question of how to generate the test, apply the test, and even analyze the test response is answered by different test techniques in different ways. Each technique has its pros and cons, which we will discuss later in this chapter.

1.1 Fault models

In order to model physical defects due to manufacturing processes a fault model is used. A variety of fault models have been proposed over the years

to model the different manifestations of defects. Tests are created depending on the fault model. A test would try to excite the fault and propagate the fault effect to one of the outputs. The response analysis is then used to see if the effect of the fault is observed. An example of a fault model is the *stuck-at* fault model [27]. In the stuck-at model, a wire is assumed to be stuck at either 0 (sa-0 fault) or 1 (sa-1 fault). The total number of single stuck at faults in a DUT with n nodes is $2n$ since each node can be stuck at 0 or 1 and only one node can be stuck at a time. A given testing technique tries to maximize the number of faults that it can detect. The effectiveness of a test technique is measured by fault coverage which is the percentage of the total faults that can be detected by the method. In this research we focus primarily on the gate level single stuck-at fault model and describe, at the end of the dissertation, how our approach can be extended to detect other faults such as path delay faults.

1.2 Fault observability and controllability

One of the biggest problems in testing is the lack of visibility of internal signals. In the case of verification, all the signals in the design are visible. This makes it possible to either check the value of an internal signal or to force a value on it to verify the functionality of the circuit. In the case of testing, the internal signals are not readily available. The only signals that are readily available are the inputs and outputs. This makes it difficult to excite some faults and also observe their effects. Hence, it is a significant challenge to

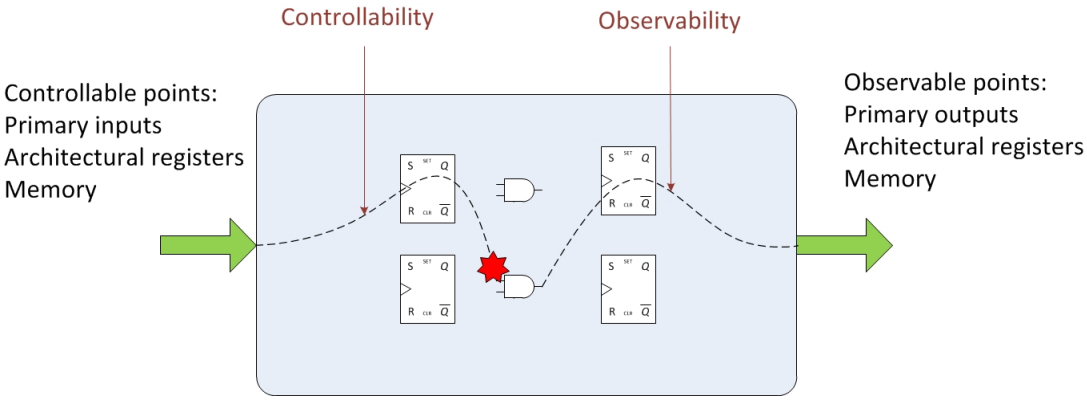

Figure 1.2: Controllability and observability

create a test for a fault that can only be applied at DUT inputs and can only be observed at DUT outputs. Controllability [21] is a measure of the ease with which a fault at a node can be excited from the DUT inputs. For example, if a node is stuck at 0 then we have to generate a 1 at the node to excite the fault. The higher the controllability for a fault at a node the more easily can the fault be excited. Observability is a measure of the ease with which a fault that has been excited at a node can be observed at the DUT outputs. Nodes that are closer to the DUT inputs are more easily controllable but are not very observable. Similarly, nodes close to the DUT outputs are more easily observable but are not easily controllable. Today complex system-on-chip (SoC) designs inherently introduce many nodes with low controllability and observability. This makes it necessary to apply a variety of approaches to achieve high fault coverage.

1.3 Conventional testing techniques

1.3.1 Automatic test pattern generation (ATPG)

The purpose of ATPG algorithms is to produce test vectors that will achieve high fault coverage for a given fault model. The test generation problem is a search problem where the objective is to find test for a given fault. Such a test when applied at the inputs of the faulty and fault-free DUT will produce a different result at the observable nodes. Thus the test, when applied in actual silicon, will uncover the presence of the fault. The observable nodes could be primary outputs, architecturally visible registers or even scanable internal registers. Many different flavors of ATPG algorithms have been proposed, the traditional algorithms being D-algorithm [58], PODEM [31] and FAN [28]. Satisfiability based algorithms have also been proposed [42]. The ATPG problem is an NP-complete problem (by reduction from the Boolean satisfiability problem which is NP-complete) and all of the algorithms rely on heuristics to give vastly improved efficiency over brute force approach. ATPG algorithms work well on combinational circuits of reasonable size and on relatively small sequential circuits. However, the ATPG problem being NP-complete implies that the algorithms suffer from the state explosion problem. A linear increase in the number of nodes results in an exponential increase in the size of the search space.

1.3.2 Scan based design for testing (DFT)

To solve the problems of observability and controllability, several modifications are done to the design just for test purposes. One such approach was first introduced by Eichelberger et al. [26], which suggested the use of scan chains in the design. Scan chains involve the use of modified flip-flops or latches connected together in a specific order like a chain. In functional mode, these sequential elements operate as normal elements. In test mode, the modified sequential elements in the chain can transfer their value to the next element in the chain. The very first element in the chain can take in value from an external input and the last element can transfer its value to a primary output. Values can be loaded (shift and launch) into internal sequential elements in the scan chain and also their values can be read out (test response). This enhances the controllability and observability of the sequential elements in the scan chain. Scan chains also alleviate the problem of state explosion in ATPG algorithms by reducing sequential ATPG problem to a combinational ATPG problem. When scan chains are used, the ATPG algorithms have to generate tests only for the combinational parts of the DUT. The test pattern can then be applied through the scan chain and the response can also be recorded using automated test equipments (ATE). During the scan-in process, since the values are shifted in one by one, there is additional overhead in terms of test application time. Compression techniques have been proposed [36] in order to compress the tests and in turn reduce test application time. Despite the use of test compression techniques, the test application time, scan test

power and the need for high speed ATEs are big drawbacks of scan based test methodologies.

1.3.3 Built-in self test (BIST)

BIST techniques solves the testing time drawback of scan chains by using special hardware for test application and response capture [35]. Additional circuitry is used on chip to generate test patterns which are applied to a sub-circuit within the chip (circuit under test, CUT). The output of CUT is then captured in a linear feedback shift register (LFSR) or a multiple input shift register (MISR) as a signature. The signature is then compared against a golden signature to check for a fault. BIST architectures comes in different flavours, BIST can be logic BIST [13] or memory BIST [67]. In logic BIST, the test vector can be applied every cycle and its response compressed simultaneously, this is called test-per-clock BIST [37] . Another approach, to capture the response into the scan chains after every test is scanned in is called test-per-scan BIST [13]. The biggest advantage of BIST is that it can be done at speed resulting in reduced test application time. Also, it does not depend on external testers, reducing equipment costs. But the overhead and additional cost appears in the form of additional on chip area and test power dissipation. The test patterns from BIST might not always give you high fault coverage. These drawbacks makes it prohibitive to use BIST on every single part of the chip.

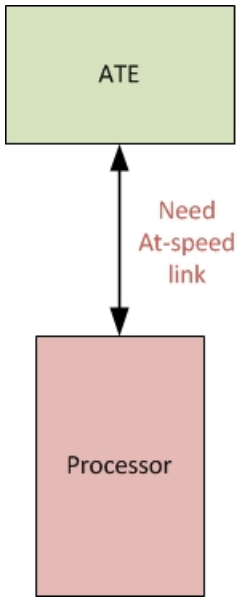

Figure 1.3: Traditional functional testing technique requiring at-speed ATE

1.3.4 Functional testing

Functional testing refers to the running of instructions as test vectors at-speed on the processor. These test vectors are typically derived manually and are targeted towards exercising specific functionality of the processor rather than specific faults. The failure of such a test would indicate the possibility of a fault. The processor under test is hooked up to an ATE which acts as the external environment as shown in Figure 1.3. The processor reads the test program through the ATE and runs it at-speed. The ATE simultaneously records the test responses which is then compared against expected values.

Functional testing was used well before the rise of DFT based approaches. Since it was not possible to get high or predictable coverage from

functional testing, DFT techniques emerged to partly address these issues. However, DFT techniques still have their drawbacks. Scan based DFT has the following shortcomings.

1. The test vectors for scan based tests are generated for a combinational circuit (DUT) without any constraints. Some of these test vectors might be non-functional and might test the circuit in states that will never be encountered. This will result in over-testing.

2. During the scan-in and scan-out process a large number of nodes are toggled and this causes high power dissipation [25].

3. Since the test patterns are shifted in through the scan chain, the test process is time consuming. Moreover the patterns are applied at a speed slower than functional mode speed. Defects that have an impact on the timing and performance of the circuit cannot be detected with vectors that run in test mode.

4. Scan sequential elements (flops and latches) are larger and slower compared to regular sequential elements. Hence an area and performance cost is involved whenever scan flops or latches are used.

BIST based DFT has the following shortcomings.

1. Even though BIST tests are applied at-speed the vectors can be non-functional resulting in over-testing as in the case of scan based DFT.

2. There is a significant area overhead for the on chip test generator and response capture logic.

3. Since the test vectors are not targeted to specific faults, a large number of vectors are required to get high fault coverage. This increases the testing time.

4. The switching activity is very high when the test patterns are being applied. This causes high power dissipation [30].

At-speed functional testing does not have any of the drawbacks of DFT techniques. All the vectors applied are functional vectors hence the issues of excessive power consumption or over-testing do not arise since the DUT will never go into illegal states [57]. Tests run at-speed so the test time is less compared to scan based DFT. On-chip area overhead isn't an issue since the only additional equipment required is the ATE. There are additional advantages to at-speed functional testing. Not all defects are captured by the fault model, some defects might change the functionality of the DUT. These defects will be caught only by at-speed functional tests. Shrinking feature sizes might cause unmodeled defects that impact the performance of the DUT. Such defects too are ideal candidates to be detected by at-speed functional vectors. These advantages make at-speed functional testing very attractive.

1.4 Software based self test (SBST)

The biggest drawback of at-speed functional testing is the need for high speed ATE, which is expensive [5]. Also it is not always possible to easily find the right set of functional tests to activate specific circuit areas. In industry, both DFT and functional testing are used actively to address coverage holes in testing.

Software based self-test also known as native-mode self test [65] [60], addresses the shortcomings of at-speed functional testing while keeping the advantages. As shown in Figure 1.4, the objective of SBST is to use the ATE only to load a program into the processor cache that will generate instruction level tests. The program will also capture the test response through the architectural registers and the data cache. The functional test might be directly loaded into the instruction cache also. SBST was inspired by BIST like approach where the test generator and response analyzer was implemented in software. This has the advantage that is no hardware overhead compared to BIST. Also we can have a low cost tester to load the test program into the instruction cache. Hence, SBST enables low cost testing with all the advantages of at-speed functional testing.

The problem of how to create instruction tests that will give high fault coverage has been the objective of SBST research. Use of random instructions as test programs is one way of creating functional tests for SBST. But random instructions give limited coverage and it is very difficult to target the uncovered faults. Manually creating tests that target these undetected faults is time

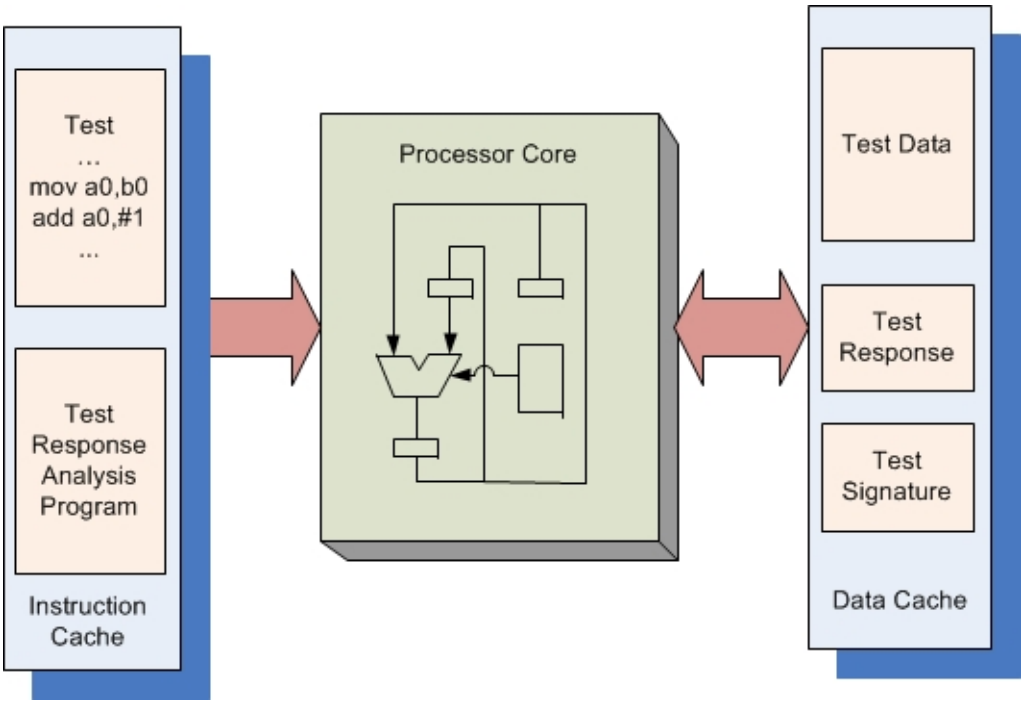

Figure 1.4: Software based self test methodology, which can use slower ATE equipment

consuming and laborious since it takes great knowledge of the architecture and gate level implementation of the DUT. Having automated tools to target these coverage holes is a necessity. Our research focuses on this issue of improving the fault coverage from functional tests. We generate instruction level tests that target faults that were left undetected by existing functional tests.

1.5 Test generation using formal methods

The problem of generating gate level tests has already been addressed by ATPG algorithms. Theoretically, these algorithms can be engineered to

generate system level functional tests by using the entire sequential DUT with the right input constraints. The input constraints for a processor constrain the generated test to be valid instructions. Most of the ATPG algorithms try to excite a fault by backward justification and then propagating the fault effect to the primary output by forward search. Since the justification and propagation steps depend on the specific fault rather than the input constraint, the search takes longer. State explosion renders ATPG based SBST impractical for designs beyond a few thousand sequential elements. ATPG algorithms that work on gate level designs are not practical for generating system level tests. The primary problem with using ATPG to generate instruction-level tests is that it is currently not easy to specify constraints on the input vectors (for example, corresponding to legal instructions).

In search of a scalable SBST method, we turn to formal methods also known as formal verification. Formal verification typically refers to the automated or semi-automated methods of checking if a design meets specification. Both the specifications and the design are captured using formal languages that may be the same or different. Then an automated or semi-automated engine checks if the design matches the specifications. Several advances have been made in the area of high level formal verification which have shown to be more scalable than previous Boolean level techniques [39]. These high level formal methods work at the RT level.

Instead of working with gate level algorithms directly, we shift the test generation problem to the RT level. This gives us several advantages. Firstly,

we can use model checking tools and the advances that are made in improving their efficiency and scalability. Our methodology can be used with the state of the art model checking tools without having to make any modifications to the flow. Secondly, working at the RTL gives us access to different types of design semantics and structures that can be used to reduce the complexity of the design and speed up test generation. These kinds of reductions would not be possible if we were working with gate level models. Finally, faster solvers are available at the RTL than at the gate level.

In our approach we first show how a gate level stuck at model can be mapped to the RTL level. This is key step since the rest of our algorithms depend on working with RTL models. Then we capture the controllability and observability as a verification property. This property is then negated and then given to a bounded model checker (BMC). The BMC takes a property and bound as the inputs and checks if the property holds on the design within the given bound. We also constrain the inputs of the design to valid instructions. A typical ATPG tool would not be able to seamlessly take in input constraints since it starts from a fault and works backwards. So a ATPG tool would generate a vector and then check it against the constraints. Verification tools seamlessly take in input constraints and perform the search based on the constraints unlike ATPG tools. The BMC tool generates a counter-example if a property does not hold. In our case with the ISA based input constraints and way we synthesize the property, the counter-example would correspond to a functionally valid test.

We further achieve scalability of our basic approach by performing reductions based on RTL semantics. These are called under-approximations, which are of two types: bit width reduction and data-path operator approximation. We analyze the RTL and remove design behaviors that are not necessary for generating a test on a given fault. Such reductions improve the efficiency of the test generation process and hence help in scaling the test generation methodology. Reductions can be made on the design also by analyzing the RTL structure. Instead of searching through all the fault propagation paths, we search through select paths at any given time. We take advantage of the ability of constraint solvers to incrementally take in the problem to be solved. At each increment we only include a subset of possible propagation paths, which reduces the state space through which the solver has to search. The choice of paths is made on the basis of RTL structure.

1.6 Contributions

The contributions of this thesis are as follows.

- We present a novel fault modeling technique where gate level faults can be captured at the RT-level. This makes it possible to use a variety of abstraction techniques at the RT-level which are not possible at the gate level. Hence, this modeling approach indirectly enables us to apply scalable test generation techniques.

- Our test generation methodology performs targeted test generation for

hard-to-detect faults. As we will see in Chapter 2, most of the existing approaches do not target specific faults. They typically use a high-level model independent of gate-level faults or they perform unconstrained gate-level ATPG. These approaches will take a long time to generate tests for hard-to-detect faults (The reason for this will be discussed in Chapter 2).

- We show that RT-level generation is highly scalable. We compare results from our methodology to the only other approach that can generate targeted functional tests for gate-level faults, viz. SAT-based ATPG.

- Our methodology can produce high functional fault coverage. The only other research work that showed high fault coverage, for a DUT similar to ours, was by Lu et al. [46]. However, a lot of micro-architectural knowledge was required from the user as inputs into the test generation tool. Our method, is completely automatic (some inputs regarding the valid instructions is necessary) and no additional insight about the microarchitecure or gate-level implementation of the DUT is required.

- We apply abstraction techniques used for RTL verification to functional test generation. We show that, with some manual inputs, these approaches can be highly scalable. Our application of these abstraction techniques is novel to test generation and from our results we see that they are very efficient for functional test generation.

1.7 Outline

In the following chapters we explain each of our techniques in detail and also present our experimental results. We survey previous work in the field of software based self test in Chapter 2. In the same chapter we discuss the issues plaguing some of previous approaches. Chapter 3 discusses some of the background ideas and some of the literature in formal methods. The ideas and terminology presented in this chapter will be used throughout this work. In Chapter 4 we discuss our RTL model checking based test generation methodology. We go into details of how a gate level fault can be mapped to the RTL model of the design under test. Synthesis of controllability and observability property by using RTL analysis is also explained. For our demonstration of our methods we used OR1200 processor as the DUT in the same chapter. We explain the under-approximation techniques for DUT abstraction in Chapter 5 and also present the experimental results. Our last technique of DUT reduction is presented in Chapter 6 along with experimental results. Finally, in Chapter 7 we discuss the advantages and disadvantages of our techniques and we also lay the ground work for future research.

Chapter 2

Software based self test: approaches and issues

Several different approaches have been taken to attack the problem of software based self test. We have classified them into three broad categories. Below we discuss each category and survey the corresponding literature.

2.1 Functionality based test generation

This category of techniques generate tests based on a high level DUT and fault model. The high level fault model is independent of gate level structural faults. Thatte et al. [65] use a graph-theoretic model of the microprocessor. The ISA of the microprocessor is used to extract the graph model and it captures the behavior of the processor in terms of register transfer functions. A functional level fault model is defined for register decoding function, instruction decoding, data storage, data transfer and data manipulation. Tests are generated based on this simple behavioral microprocessor description and fault model. Brahme et al. [17] account for more complex microprocessor execution sequences and describe a more sophisticated functional model. Goor et al. [69] extend this functional model based test generation approach to microprocessor caches. Shen et al. [61] propose a control fault model which covers register de-

coding faults, instruction decoding faults and instruction execution sequence faults. This control fault model is defined for a "kernel" microprocessor with read/write instructions. The kernel is then verified separately using checking experiments.

A different approach was proposed by Shen et al. [60] where they use random instruction sequences as functional tests. They classify the instructions in the ISA into observation sequences and control sequences. Observation sequences are instructions that propagate register values. Control sequences are instructions that manipulate register values. Random instruction based tests are then generated by systematically choosing a combination of control sequences and observation sequences. This methodology, which is akin to a software BIST, was used on an industrial design by Parvathala et al. [53]. They compared the fault coverage from these tests against DFT based techniques. They found that faults that escaped normal test flow were detected by this methodology.

These methods have fault models that are independent of the implementation and they do not have a good correlation with manufacturing defects. Hence it is not possible to achieve high structural fault coverage with these methods. To address this issue, techniques have been proposed where the high level test generation targets the gate level faults. Kranitis et al. [38] use the RT level description of the processor. Their methodology generates deterministic test patters to excite the gate level faults in RT level components such as ALUs, registers, multiplexers etc. The relationship between the instructions

and processor components is manually extracted. A high level test generation approach based on Satisfiability Module Theories (SMT) solvers is proposed by Alizadeh et al. [12]. Their tests are again based on a high level behavioral fault model and not specifically targeted at gate level faults. Corno et al. [24] generate tests using evolutionary algorithms. The fitness criteria used during the test generation process is the fault coverage attained. Lu et al. [46] present a methodology based on random program generation to create software tests that give high fault coverage. However, the test program generator depends on additional information about the instruction set architecture and micro-architecture.

2.2 Constraint extraction based approach

In constraint based approach, test generation is performed by constraining the module which has the specific gate level fault. The constraints are extracted by analyzing the logic surrounding the module. The constraints should be generic enough to justify the fault and then propagate the module responses, and at the same time should not allow non-functional patterns. Such approaches are highly scalable and provide high coverage.

Tupuri et al. [66] extract functional constraints for a module under test. They refer to these functional constraints as "virtual" circuit constraints. An ATPG algorithm is then used to generate tests for faults in the module with the virtual circuit constraints as the input-output environment of the module. A similar approach of extracting the environment constraints was proposed by

Vedula et al. [68]. They use program slicing to remove the RT-level statements that are irrelevant to the module under test. The sliced HDL code is used as the virtual circuit constraint and an ATPG tool is used for test generation.

In both approaches the ATPG tool works on a significantly reduced design and hence is able to generate efficient tests. However, extracting such functional constraints for any given module in modern microprocessors is a challenging problem. Chen et al. [22] circumvent this issue by extracting functional constraints defined by specific instruction templates. Instruction templates are program templates created on the basis of the ISA. Templates are initially ranked on the basis of a simulation, templates that give high controllability and observability for the module under test are ranked higher. Then a simulation is performed by assigning random values to the settable fields of the instruction template. Based on a regression analysis of the simulation values seen at the input and outputs of the module under test, input and output mapping functions are extracted. Virtual circuit constraints are extracted from these mapping functions and finally ATPG based test generation is performed. The tests generated are mapped back to the instruction templates. The quality of test coverage depends on the instruction templates, and writing extensive instruction templates requires knowledge of the instruction set architecture.

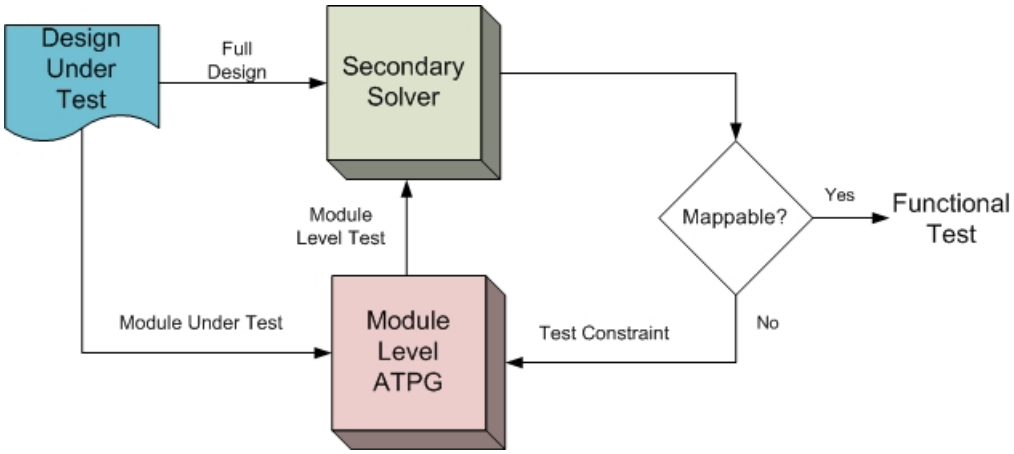

Figure 2.1: Mapping based approach for software based self test

2.3 Pre-computed test mapping approach

The most popular approach is the pre-computed test based approach, in which gate level ATPG is used to generate module level tests, and then an RTL based approach is used to justify the module level tests and propagate the responses to the primary outputs. This divide and conquer approach makes it possible to generate tests for complex designs. Special knowledge of module implementation or the architecture isn't necessary. Specialized solvers can be used at the module level and at the DUT level. Figure 2.1 shows a flowchart for the mapping based approach.

Murray et al. [52] present a mapping approach for acyclic designs. Module level tests are contained in test packages. System level tests are then generated by symbolically manipulating the test packages. The module level responses are propagated along circuit paths. They then extract the tests from the symbolic responses. Roy et al. [59] use a data flow descriptions

for generating tests. ATPG is used to generate tests for stuck-at faults for combinational circuit components. Then, the data-path (from the data flow descriptions of the circuit) is used to propagate the fault effect from combinational component output to the DUT output. A similar algorithm is used to justify the module level tests to the component input. Sequential propagation and justification of signal values is carried out recursively. An automatic test knowledge extraction tool (ATKET) is presented by Vishakantaiah et al. [70]. The tool builds a data structure from the RT-level description for each module that they refer to as module operation tree. Using the information stored in the operation the search space is pruned for test justification and error propagation. Bhatia et al. [14] incorporate test generation into behavioral synthesis. Their methodology works for designs that have separate controller and data path logic. The system level tests generated are based on precomputed module level tests. Since design synthesis is a necessary part of the process, their work can be considered as design for testability. Assignment decision diagrams (ADDs) are used by Lingappan et al. [45]. Input/output propagation rules are used to abstract out the components of the ADD. The justification/propagation requirements of the module level tests are then captures as Boolean implications. Satisfiability (SAT) solvers are then used to solve the implication to get the system level test. Zhang et al. [73] use a similar approach with ADDs but apply ATPG based techniques to solve Boolean implications.

Gurumurthy et al. [32] [33] use ATPG to generate module level tests. These module level tests are then mapped to the system level using a bounded

model checker. The controllability and observability constraints of the module level test is captured using linear temporal logic (LTL). The LTL properties for a module level test are synthesized such that a witness to the property would constitute a system level test which justifies and propagates the module level test. This approach makes use of existing tools for ATPG and for verification, so off the shelf state of the art tools can be used.

The mapping methodology has the following disadvantages.

1. The ATPG performed is unconstrained, hence a module level test may not always be mapped to the system level.

2. Convergence might be slow if high coverage is required since the hard to detect faults are not targeted. There is no guarantee that a specific fault that can have a gate level test will have a system level mapping.

3. For scalability manual abstractions such as removing multiplier would be necessary to do the system level mapping.

We address these shortcomings in our research. The main objective of our research is "the holy grail of testing", viz., functional vectors with high structural coverage.

Chapter 3

Formal Methods

3.1 Introduction

Formal methods are primarily concerned with checking if a logic formula is valid. An example of a logic formula is a Boolean formula. In formal verification the problem of checking a specification against a design is cast as a formula validity problem. There are two fundamental approaches to check the validity of a formula [40].

- Deduction based approach, where deductive reasoning using axioms and inference rules check the validity of a formula. Theorem provers like ACL2 [2] use this kind of reasoning.
- Enumeration based approach, where all the possible candidate solutions of the formula are enumerated and checked. Model checking [23] uses this kind of reasoning.

The enumeration based proof has the advantage that in the case the formula is not valid, it has the capability to generate candidate instances for which the formula is not satisifiable. We can use this mechanism to generate tests. Deduction based approach typically is able to give a yes or no answer

without producing candidate instances of invalidity. User insight is required to analyze the results of a failed deduction. Hence in our research we use model checking tools and their capability to generate counter-examples.

3.2 Model checking

As mentioned earlier, model checking [23] is a technique that can be used to check if the design meets specification. Specifications are captured using temporal logic which is capable of expressing time based relationship between states in a finite state machine. The very first temporal logic proposed for model checking was computational tree logic (CTL) [23], and then linear temporal logic (LTL) [54] also gained in popularity (even though LTL was proposed before CTL). Today in the industry, system verilog assertions (SVA) [11] is very popular method of specifying assertions on verilog designs.

3.2.1 Unbounded vs bounded model checking

Unbounded model checking means checking properties against the design without any time bounds. If the unbounded model checking returns true then the property always holds true on the design. Usually model checking almost always refers to unbounded model checking. These types of checkers start from the initial state and recursively checks the reachable states for a failure.

Bounded model checking (BMC) [15] checks properties only within a user specified time bound. If the BMC returns true then it means that the

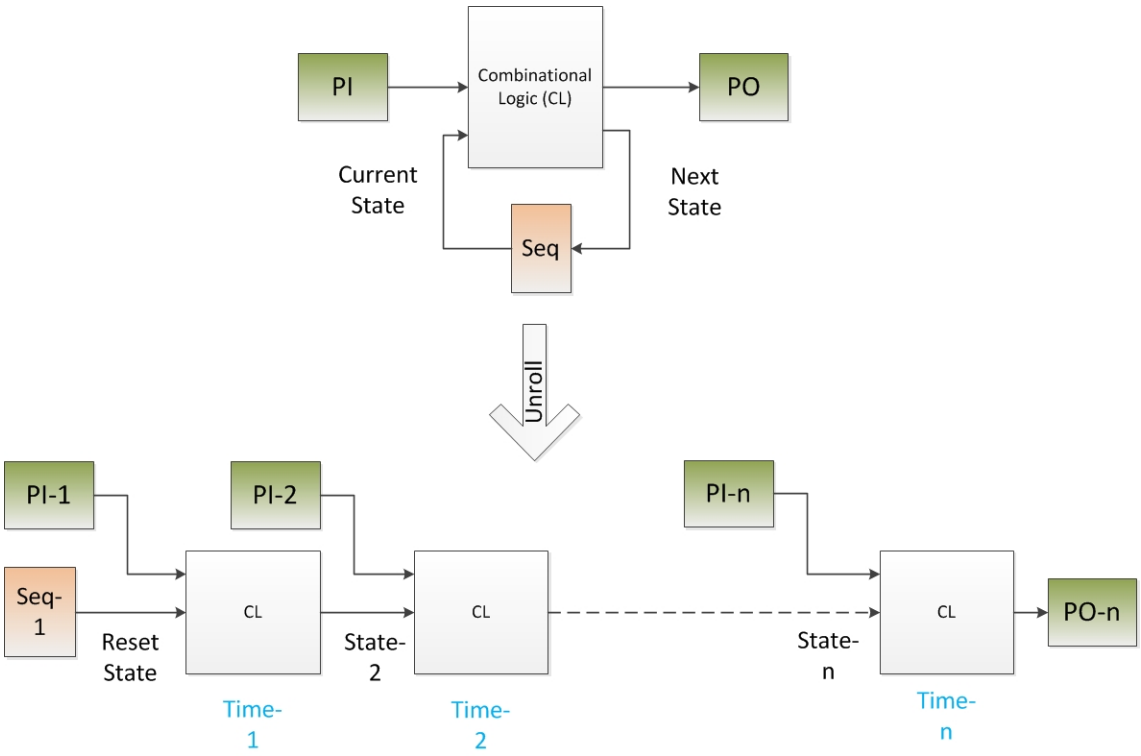

Figure 3.1: Unrolling a DUT for a bound of n cycles

property only holds within the bound time. It is possible that the property fails outside the time bound. Hence bounded model checking is not a complete check. As shown in Figure 3.1, BMC for a sequential design involves unrolling the circuit n times (for a bound of n), each unrolled instantiation corresponds to one time frame. The property to be verified is also translated for the bound of n.

Unbounded model checking is not as scalable as BMC since the search space can be quite large for the entire system. On industrial designs an unbounded check can run for days without returning an answer. In the case of

BMC the user has control over how deep the search needs to be performed, hence is much more scalable than unbounded model checking. BMC is typically used to find bugs in a design since it cannot be used as a complete verification method. If a property does not hold true then the BMC returns a counter-example that caused the property to fail. We use this feature of BMC in our research to generate tests.

3.2.2 Properties

Three types of properties are used in design verification [63].

1. Safety: these properties are used to specify the good reachable states in the design. If a state in which the safety property does not hold is reachable, then the verification fails.

2. Fairness: these properties are of the form, a state that can be infinitely often enabled should be reachable infinitely often.

3. Liveness: these properties specify that a state should be eventually reachable.

We use safety properties for test generation. Our choice of safety properties is explained in the next chapter. The property that an error from a fault should reach one of the observable points in the DUT is the observability property. This property is negated and given to the BMC. If the fault is observable then we have a counter-example to the negated property. This counter-example will be our test.

Chapter 4

Functional Test Generation for Hard to Detect Stuck-At Faults using RTL Model Checking

4.1 Introduction

In this research we present a generic approach for generating functional tests for gate level stuck-at faults. Our approach is similar to gate level ATPG in that we target specific faults but is more scalable since we generate tests based on the RTL. Our technique does not supplant existing functional test generation methods but complements them since it generates tests for the remaining hard to detect faults that were not detected using an existing technique. We model the gate level stuck-at fault in RTL using Boolean difference. The fault controllability and observability conditions are then captured as an LTL property [47]. A RTL Bounded Model Checker (BMC) is then used to find a counter-example to the negation of this property. The counter-example would contain the instruction level test for the gate level stuck-at fault. If a counter-example is not produced then the fault is termed untestable. Scalability is achieved by converting the RTL design and LTL property into a bit-vector BMC problem, which is then solved using an SMT solver. We further achieve scaling by extracting a structural observability property for the stuck-at fault using the Control Data Flow Graph (CDFG) of the RTL. This

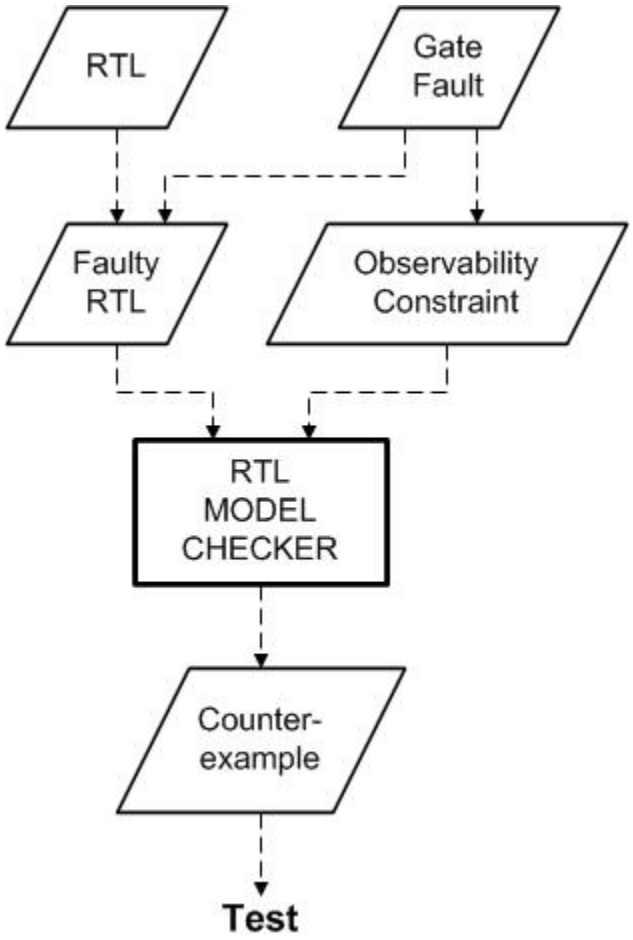

Figure 4.1: RTL test generation methodology for gate level faults

affords us scaling by constraining the state space of the search and structurally reducing the size of the BMC problem.

A flow chart of the methedology is shown in Figure 4.1. The contributions of this research are as follows.

1. We introduce a test generation technique for gate level faults using RTL Bounded Model Checking.

2. Our technique is generic enough so that it can be applied to any RTL based design.

3. We provide a theoretical framework to check the observability and controllability of a stuck-at fault as an RTL Model Checking problem.

4. We leverage advances in RTL level Formal Methods like SMT solvers to solve the test generation problem. We also present a technique to further leverage the use of RTL by optimizing the BMC formulation for test generation.

4.2 Preliminaries

4.2.1 Boolean difference

Boolean difference is used to express the observability of a fault at a given observable point. Applying Boolean difference to generate tests for stuck-at faults using Satisfiability (SAT) solvers was first introduced by Larrabee [43]. Given two Boolean functions O and O' which have the same inputs $x_1, x_2..., x_n$, the Boolean difference between the two functions is expressed as

$$O_{bd} = O(x_1, x_2..., x_n) \oplus O'(x_1, x_2..., x_n)$$

If O represents the output of the fault-free combinational circuit and O' represents the output of the corresponding faulty circuit then O_{bd} represents all possible input combination under which F and F' differ. If $O_{bd} = 0$ then the fault is not detectable under any input values. Hence Boolean difference

can be used to capture the effect of a fault at the output of a circuit. The output of the faulty circuit can also be expressed as

$$O' = O(x_1, x_2..., x_n) \oplus O_{bd}(x_1, x_2..., x_n)$$

4.2.2 Model Checking

Given a Finite State Machine M and temporal property P, a Model Checker verifies if the property P is true of the model M or it false. We use temporal properties expressed in LTL [47]. The LTL operator X represents the *next state* operator and $X(P)$ means that the property P should be true in the state after the current state. F represents the *future* operator and $F(P)$ means that the property P should hold in some future state after the current state. G represents the *global* operator and $G(P)$ means that the property P should be true in all states, i.e, the current state and the ones that appear in the future.

Model checkers come in different flavors. Some approaches are tuned to prove a property and some are designed to falsify a property. Bounded Model Checking [15] is one such approach that is more suitable to falsify a property. A Bounded Model Checker checks if a property P holds for the model M within a given time bound n. A BMC based on SMT (or SAT) solvers works as follows.

Let $M = (S_0, I, PO, R, RF, T)$, where S_0 is the initial state constraint, I is the set of primary inputs, PO is the set of primary outputs, R is the set of internal registers, RF is the register file and T is the set of transition rela-

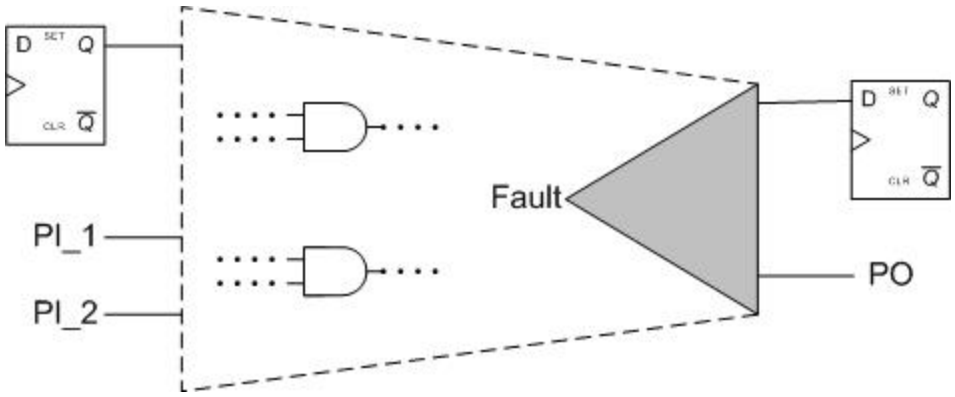

Figure 4.2: Transitive Fan-in Illustration

tionships between one register state to another. A BMC creates the following formulation of M and P

$$B = S_0 \wedge M_0 \wedge M_1 \wedge M_2 \wedge ... \wedge M_n \wedge (\neg P)$$

where $M_i = R_i \wedge I_i \wedge PO_i \wedge T_i$ is the SMT-lib (or CNF) formulation of the model M at time frame i, and $\neg P$ is the SMT-lib (or CNF) formulation of the negation of the property to be falsified. If the SMT (or SAT) solver finds that the SMT-lib (or CNF) formula B is satisfiable then the formula P does not hold true. Moreover the counter-example can be extracted from the assignments to $I_0, I_1, ..., I_n$.

If the SMT (or SAT) solver finds that the formula B is not satisfiable then we would have to rerun the BMC with a larger bound. Hence BMC is an incomplete method suited to falsification. We use the counter-example generation feature of BMC to generate tests.

4.3 Approach

4.3.1 Capturing Gate level faults in RTL

In our approach we assume that there is a one to one match between the registers in the synthesized gate level netlist and the RTL of the Design Under Test (DUT). This means that a combinational equivalence check between the netlist and RTL should be satisfied. In most designs this is true except when parts of the circuits are retimed [44]. Our method in its current form cannot be applied to those retimed parts of the circuit.

For any given stuck-at fault in the netlist we extract registers and primary outputs in its combinational output cone. For each of these registers/primary outputs we then extract the combinational circuit up to the registers and primary inputs in their input cone, i.e., the combinational circuit driving them. This is illustrated in Figure 4.2, where the area shaded in gray is the fan-out of the fault and the circuit enclosed by the dashed line is the transitive fan-in. In essence we are extracting the outputs affected by the given stuck-at fault and the transitive fan-in of the combinational circuit in which the stuck-at fault exists.

Let $o_1, o_2, ..., o_m$ be the registers/primary outputs in the combinational output cone of the stuck-at fault. Let the combinational circuit driving these outputs be represented by the Boolean function

$$o_j = f_j(i_{j,1}, i_{j,2}, ..., i_{j,l})$$

where $i_{j,1}, i_{j,2}, ..., i_{j,l}$ are the registers/primary inputs of the DUT that drive

the combinational input cone of o_j.

We compute the corresponding Boolean function for o_j with the fault inserted in the combinational input cone. Let this Boolean function for the faulty circuit be represented as

$$o_j^f = f_j^f(i_{j,1}^f, i_{j,2}^f, ...i_{j,l}^f)$$

The fault is propagated to the output o_j^f when its value differs from the corresponding value in the fault-free circuit, i.e., o_j. This can be expressed as the following Boolean difference.

$$fault_{o_j^f} = f_j(i_{j,1}^f, i_{j,2}^f, ...i_{j,l}^f) \oplus f_j^f(i_{j,1}^f, i_{j,2}^f, ...i_{j,l}^f)$$

Whenever the **fault condition** $fault_{o_j^f}$ is satisfied the fault is activated at o_j^f and its value differs from the fault-free value. $o_1^f, o_2^f, ..., o_m^f$ are referred to as **fault activation points**, since the fault effect is seen at these points. Note that all the signals $o_j^f, i_{j,1}^f, i_{j,2}^f, ...i_{j,l}^f$ exist in the RTL because of the combinational equivalence assumption.

The faulty RTL model M^f is constructed from the fault-free RTL model M by inserting muxes at the fault activation points (see example later in this section). All occurrences of o_j in the LHS of assignments (continuous/always block assignments) in M are replaced with $o_j'^f$. $o_j'^f$ contains the fault-free value in M^f. Occurrences of o_j in the RHS of assignments in M are replaced with o_j^f in M^f. When fault condition $fault_{o_j^f}$ is false $o_j^f = o_j'^f$, and when the fault condition is true the fault is activated and $o_j^f = \neg o_j'^f$. Hence the

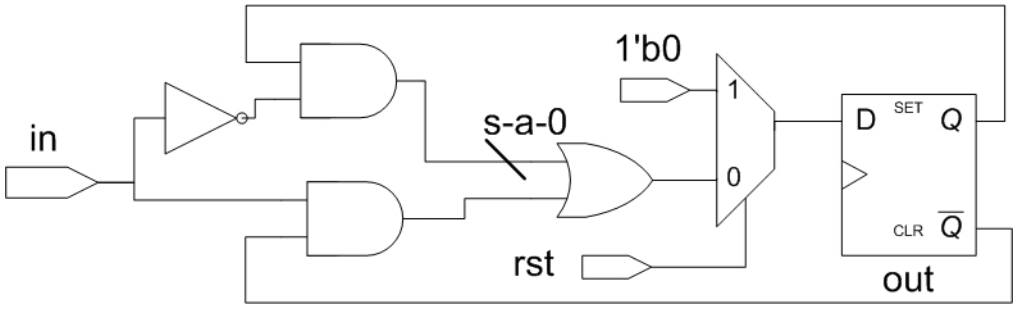

Figure 4.3: Stuck-at Fault Example

fault condition determines when to inject the fault into the RTL. The faulty machine $M^f = (S_0, I, PO^f, R^f, RF^f, T^f)$ is a copy of the fault-free machine with the fault injection logic added to T^f.

As an example consider the Verilog RTL code below.

$always$ @($posedge\ clk$)

$if\ (rst)\ out <= 1'b0;$

$else\ out <= in \oplus out;$

The corresponding gate level netlist and the stuck-at fault is shown in Figure 4.3. The fault condition is $fault_{out} = (\neg rst \wedge \neg in \wedge out)$.

The faulty RTL is as shown below. The signals with the superscript f indicate that these are signals in the faulty RTL.

$always$ @($posedge\ clk$)

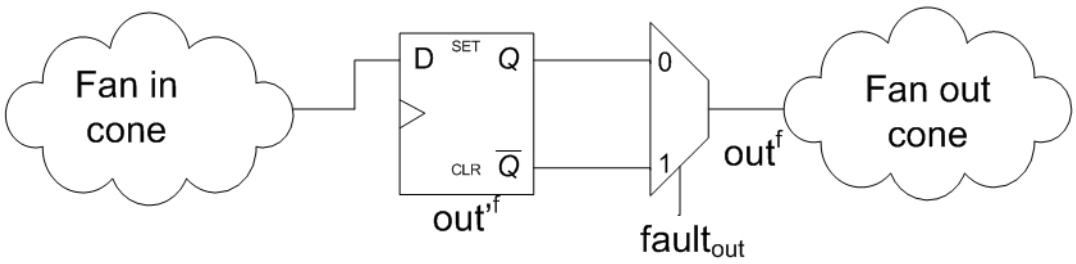

Figure 4.4: Fault Injection Mux Example

$$if\ (rst)\ out'^f <= 1'b0;$$

$$else\ out'^f <= in^f \oplus out^f;$$

$$assign\ fault_{out} = (\neg rst \wedge \neg in \wedge out'^f);$$

$$assign\ out^f = (fault_{out}?\neg out'^f : out'^f);$$

The circuit representation of the Faulty RTL with the fault injection mux is shown in Figure 4.4.

Algorithm 4.1 gives the general outline for injecting faults into the RTL. Given the RTL (DUT_{RTL}), a gate level module ($M_{netlist}$) and a fault (F_g, F_p, F_t) in the gate level module, it creates a faulty RTL ($DUT_{FaultyRTL}$) corresponding to the fault. At step 1 of the algorithm, we extract the faulty version of the gate level netlist ($M_{FaulyNetlist}$) using the gate (F_g), pin (F_p) and stuck-at fault type (F_t). Then we identify all the primary outputs and flops in the combinational fan-out cone of the fault, this is represented as $S_{PO,flops}$. For each of the signals

($OuputSig$) in $S_{PO,flops}$, we extract the ROBDD (Reduced Ordered Binary Decision Diagram [20]) corresponding to the combinational fan-in cone, which is represented as $BDD_{OutputSig}$. We also extract the corresponding faulty ROBDD from $M_{FaultyNetlist}$, which is represented as $FAULTYBDD_{OutputSig}$. We then extract the ROBDD for the fault condition ($BDD_{FaultCond}$) by xoring $BDD_{OutputSig}$ and $FAULTYBDD_{OutputSig}$. The SOP (Sum of Products) is extracted from $BDD_{FaultCond}$, represented as $SOP_{FaultCond}$. Finally, we insert a mux at $OuputSig$ in $DUT_{FaultyRTL}$ using $SOP_{FaultCond}$ to get the faulty RTL.

Algorithm 4.1 Fault injection

Input: DUT_{RTL}, $M_{netlist}$, (F_g, F_p, F_t)
Output: $DUT_{FaultyRTL}$

1: $M_{FaulyNetlist}$ = inject_fault($M_{netlist}$, (F_g, F_p, F_t));
2: $DUT_{FaultyRTL} = DUT_{RTL}$;
3: $S_{PO,flops}$ = extract_combo_primary_outputs_flops($M_{netlist}$,F_g);
4: **for all** $S_{PO,flops}$ **do**
5: $BDD_{OutputSig}$ = extract_combo_fan_in($M_{netlist}$,$OuputSig$);
6: $FAULTYBDD_{OutputSig}$ = extract_combo_fan_in($M_{FaultyNetlist}$,$OuputSig$);
7: $BDD_{FaultCond}$ = BDD_XOR($BDD_{OutputSig}$,$FAULTYBDD_{OutputSig}$);
8: $SOP_{FaultCond}$ = extract_SOP($BDD_{faultCond}$);
9: $DUT_{FaultyRTL}$ = insert_mux($DUT_{FaultyRTL}$, $SOP_{FaultCond}$);
10: **end for**
11: **return** $DUT_{FaultyRTL}$

4.3.2 Test Generation Using Model Checking

For identifying the conditions for observability of the fault we construct a product machine $M \times M^f$, where

$$M \times M^f = (S_0, I, PO \cup PO^f, R \cup R^f, RF \cup RF^f, T \cup T^f)$$

In the product machine the primary inputs are shared between M and M^f, but every other signal has two copies, one in M and another in M^f.

The fault controllability LTL property is given by

$$C_{fault} = F(fault_{o_1^f} \vee fault_{o_2^f} \vee ... \vee fault_{o_m^f})$$

which means for the fault to be activated, any one of the fault conditions should be true in the future. If the model checker proves that the property C_{fault} does not hold on $M \times M^f$ then the fault is not controllable. Hence model checking to prove $\neg C_{fault}$ can serve as a quick check to prune out uncontrollable faults.

The fault observability LTL property is given by

$$O_{fault} = (F(\bigvee_{\forall i} BD_{PO_i}) \vee F(\bigvee_{\forall k} G(BD_{RF_k}))))$$

where $BD_{PO_i} = (PO_i \oplus PO_i^f)$, which represents a difference in values at the ith primary output of the good and faulty machine. And $BD_{RF_k} = (RF_k \oplus RF_k^f)$, which represents a difference in values at the kth register file of the good and faulty machine. The first part of O_{fault} captures the condition that the fault effect should be propagated to any one of the primary outputs in the future. The second part captures the condition that once a fault effect is propagated to a register file it remains there, which is specified by the global LTL operator. This is done from a test generation perspective. If the model checker generates a counter-example for the property $\neg O_{fault}$ and it shows that a fault effect can be propagated to a register file then we would like the fault effect to be stored there till the end of the counter-example. The next instruction we append to

the counter-example would be a read from the register file to which the fault effect was propagated. This would ensure that the fault effect is propagated to a primary output from the register file.

If the model checker can prove that O_{fault} does not hold on $M \times M^f$ then we can conclude that the fault is not observable. Proving that O_{fault} does not hold is harder to do that to prove that C_{fault} does not hold. This is because the search space of the Observability property is much larger than the search space of the controllability property.

We ask a Bounded Model Checker(BMC) to falsify the property $\neg O_{fault}$. The initial state S_0 of $M \times M^f$ is chosen to be some valid functional state for the internal registers and the register file is left unconstrained. The inputs are restricted to be valid instructions so that we get functionally valid counter-examples. For a given fault the minimum bound that should be given to the BMC is $(I_{min} + O_{min})$ where I_{min} is the smallest number of cycles for a value at a primary input to reach any one of the fault activation points and O_{min} is the smallest number of cycles required for a value at the activation point to reach any one of the primary outputs. If the BMC generates a counter-example to $\neg O_{fault}$, then that counter-example is a functional test to activate and detect the stuck-at fault. If the BMC is not able to generate a counter-example then we try with a larger bound.

4.4 Observability Property using CDFG

The functional test generation approach mentioned in the previous section has two main disadvantages. Firstly, for a given bound there are two unrolled copies of the RTL, one is the fault-free version and the other is the faulty version. If we can reduce the size of unrolled product machine $M \times M^f$ we can reduce the search time for finding a test. Secondly, since there are no additional constraints other than the instruction constraints at the input, the solver underlying the BMC should do all the work of finding a valid propagation path for the fault effect. This increases the search space for the solver. The search space can be reduced if we add additional path propagation constraints for a given fault. Yang et al. [72] show that supplying intermediate conditions to a BMC engine helps falsify a property faster. In this section we solve the above two problems by extracting a structural observability property based on the CDFG of the RTL. This property helps reduce the size of unrolled $M \times M^f$ given to BMC and also provides intermediate constraints for the propagation path of the fault.

4.4.1 Structural Dependency Graph

To create the observability property for a fault we extract a Dependency Graph for the fault from the CDFG of the DUT. Let the dependency graph be $D_g = (V, E)$, where V is the set of vertices and E is the set of edges (v_i, v_j) where $v_i, v_j \in V$. Each of the vertices in D_g corresponds to signals(wires, registers) in the RTL. A signal b is said to be data dependent on signal a if a

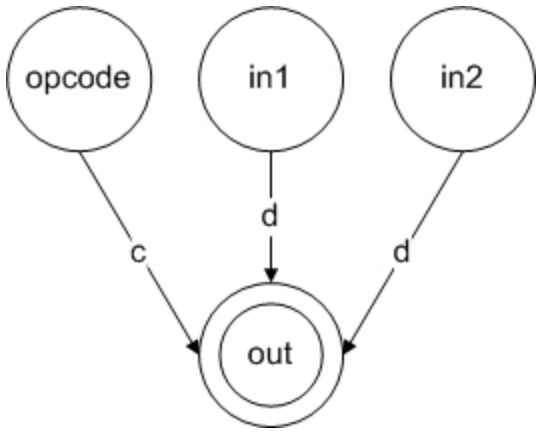

Figure 4.5: Example of a control and data dependency graph

appears on the RHS of an assignment to b. b is said to be control dependent on a if a appears in a conditional statement (examples are *if* and *case* statements) that makes an assignment to b. An edge (v_i, v_j) exists in D_g if the signal corresponding to v_j is data or control dependent on the signal corresponding to v_i.

Consider the verilog code shown below, the dependency graph for this code is given in Figure 4.5. The edge labeled 'c' captures the control dependency and the edge labled 'd' captures the data dependency.

```
always @(posedge clk)
  case (opcode)
    2'b0: out <= in1 + in2;
    2'b1: out <= in1 - in2;
    2'b2: out <= in1 * in2;
    2'b3: out <= in1 << in2;
  endcase
```

D_g for a given fault is created by performing a breadth first search

starting at the fault activation points. Any signal that is control or data dependent on any of the fault activation points is added to D_g. The graph grows by continuing the breadth first search on the newly added vertices. We stop the breadth first search when we reach a signal that is a primary output or a register file. Hence the dependency graph D_g represents all the possible paths for the fault effect to propagate from the fault activation points to the observable points (primary outputs and register file).

4.4.2 Observability Property

The structural observability property is now constructed from the Dependency Graph D_g. Every vertex v in D_g will have an LTL property P_v associated with it. P_v will depend on the type of vertex v.

1. If v corresponds to some primary output signal PO_i, then $P_v = BD_{PO_i}$, where $BD_{PO_i} = (PO_i \oplus PO_i^f)$

2. If v corresponds to some register file RF_i, then $P_v = G(BD_{RF_i})$

3. If v corresponds to some internal wire IW_i, with $w_0, w_1, ..., w_n$ as the successors of v in D_g, then $P_v = (BD_{IW_i} \wedge (LTL\_Op(P_{w_0}) \vee LTL\_Op(P_{w_2}) \vee ... \vee LTL\_Op(P_{w_n})))$, where $LTL\_Op(P_{w_k}) = X(P_{w_k})$ if successor w_k corresponds to a register, else $LTL\_Op(P_{w_k}) = P_{w_k}$

4. If v corresponds to some internal register R_i, with $w_0, w_1, ..., w_n$ as the successors of v in D_g, then $P_v = (BD_{R_i} \wedge (LTL\_Op(P_{w_0}) \vee LTL\_Op(P_{w_2}) \vee ... \vee LTL\_Op(P_{w_n}) \vee X(P_v)))$, where $LTL\_Op(P_{w_k})$ is as defined in (3).

Property (1) above ensures that the fault effect is observed at the primary output in the state in which it is evaluated. Property (2) ensures that the fault effect remains in the register file till the end of the counter-example as discussed in Section 4.3.2. Property (3) captures the case when a fault effect is seen at in internal wire, then the fault effect should be propagated to any one of its successors. If the successor is a register then the fault effect is propagated in the next time frame. If the successor is not a register then the fault effect is propagated in the current time frame. Property (4) is similar to property (3) in that it ensures the fault effect is propagated to successors. But since this property is specific to vertices corresponding to internal registers, it has the choice of propagating the fault effect in the next time frame as well. Hence this property is recursive.

For any given vertex v, if P_v is true then it implies that the fault effect was seen at the signal corresponding to v and it was propagated to one of the observable points. If $v_1, v_2, ..., v_n$ are vertices in D_g corresponding to fault activation points then the final observability property is given by

$$O_{fault} =$$

$$F((P_{v_1} \vee P_{v_2} \vee ... \vee P_{v_n}) \wedge ((\bigvee_{\forall i} BD_{PO_i}) \vee (\bigvee_{\forall k} G(BD_{RF_k}))))$$

O_{fault} specifies that in some future time the fault will be activated at any one of the fault activation points and will be propagated to one of the observable points. Hence a counter-example to the property $\neg O_{fault}$ will serve

as functional test for fault activation and detection. This property restricts the search space to only the possible propagation paths for the backend solver of the BMC.

Consider the verilog code below.

```
always @(posedge clk)
  sum <= PI | sum;

assign O1 = sum & t1;

always @(posedge clk)
  t3 <= sum & t2;

always @(posedge clk)
  if (t3)
    O2 <= a || b;
```

The dependency graph corresponding to this code is given in Figure 4.6. Let us assume that $D/\overline{D}$ is in the signal sum and we need to propagate it to one of the observable outputs *O1* or *O2*. We create the following intermediate and final observability properties.

$BD_s = s \oplus s^f$

$P_{O1} = BD_{O1}$

$P_{O2} = BD_{O2}$

$P_{t3} = (BD_{t3} \wedge (X(P_{O2} \vee P_{t3})))$

$P_{sum} = (BD_{sum} \wedge (P_{O1} \vee X(P_{t3} \vee P_{sum})))$

Final observability Property :

$F(P_{sum} \wedge (P_{O1} \vee P_{O2}))$

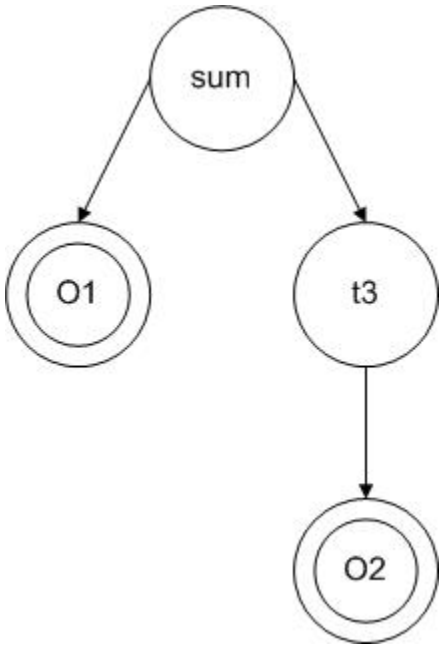

Figure 4.6: Control and data dependency graph of verilog example

4.4.3 Structural Reduction

The Dependency Graph D_g and the structural observability property O_{fault} affords us further reduction in the size of the unrolled product machine $M \times M^f$.

4.4.3.1 Reducing Duplicated Signals

In Section 4.3.2 we saw that while constructing $M \times M^f$ every signal in the design other than primary inputs is duplicated, i.e., there is a fault-free and a faulty copy. Since D_g exactly keeps track of those signals that see the fault effect, a signal that does not appear in D_g need not be duplicated since the fault effect will not be propagated to that signal. Hence we maintain a

single fault-free version of such signals. This reduces the number of duplicated signals in $M \times M^f$.

Let s_{min} be the minimum number of cycles required to propagate a fault effect from any of the fault activation points to the signal s, where s is in D_g. s_{min} can be computed as the minimum of the number of registers encountered on any path from any of the fault activation points to s. Since signal s does not see the fault effect till time frame s_{min}, we can maintain a single fault-free copy of s for each time frame less than s_{min}. This further reduces the number of duplicated signals in the unrolled product machine $M \times M^f$ for the bounded model check.

4.4.3.2 Bounded Cone of Influence Reduction

Bounded Cone of Influence Reduction [16] removes irrelevant signals and behaviors from the unrolled product machine $M \times M^f$. This is achieved by identifying those signals that do not have direct or indirect impact on the structural observability property O_{fault}. Both the irrelevant signal and the assignment to those signals are removed from the unrolled product machine. This reduction is possible because the observability property is structural and would not have been possible with observability property given in Section 4.3.2. Both reductions discussed reduce the size of formula given to the backend solver and hence help in further reducing the running time of the BMC.

4.5 Experimental Results

4.5.1 OR1200 Processor

We carried out experiments on an Intel quad-core 2.5 GHz server with 32 GB of RAM. We used OR1200 processor [8] as the DUT. OR1200 processor is an open source RISC processor with a 5 stage pipeline. The verilog source code and the instruction set manual is available at [8]. The processor has a 5 stage pipeline with separate data and instruction caches. It has all the basic pipeline units like instruction fetch, instruction decode, execute, write-back and memory access. In addition it has an exception handling unit. It has 32, 32-bit general purpose registers. The synthesized netlist of the processor has approximately 3.4K state elements and roughly 37K gates.

In order to get valid instructions in the test generation process, we had to constrain some of the inputs of the OR1200 processor. This is discussed in [32]. We used the below verilog constraint, similar to that introduced in [32].

```
always
  begin
  // rst and stall deactivation
  icpu_err_i = 0; //for deactivating fetch stalls
  du_stall = 0;
  dcpu_rty_i = 0;
  rst = 0; // disable reset

  ...

  if (!VALID_INSTR)
    assert VALID_INSTR_PR: (1'b0);
  end
```

```
always (*)
  begin
    if ({icpu_dat_i[31], icpu_dat_i[30], icpu_dat_i[29],
     icpu_dat_i[28], icpu_dat_i[27], icpu_dat_i[26]} ==
     `OR1200_OR32_ADDI ) { // integer addition
      VALID_INSTR = 1'b0;
    } else if {
      ... // Constraints for other uops here
    } else { // None of the instructions match
      VALID_INSTR = 1'b0;
    }
  end
```

The first part of the constraints corresponds to disabling reset and other conditions such as a fetch stall due to cache miss. The second part captures the constraint that the generated instructions should be valid. *VALID_INSTR* is true only if the value in the signal that fetches the instruction from the instruction cache, *icpu_dat_i*, is a valid opcode. We add a condition that the assertion *VALID_INSTR_PR* should never be fired when generating a test. We had to manually look up the OR1200 processor ISA and create the valid instructions constraint.

4.5.2 Identifying Hard-to-detect Faults

We considered faults only in the control part of the circuit since data-path elements like adders and multipliers are easily controllable and observable and can be effectively tested by random instructions. Stuck-at faults were injected in all modules of OR1200 processor netlist except the ALU and then

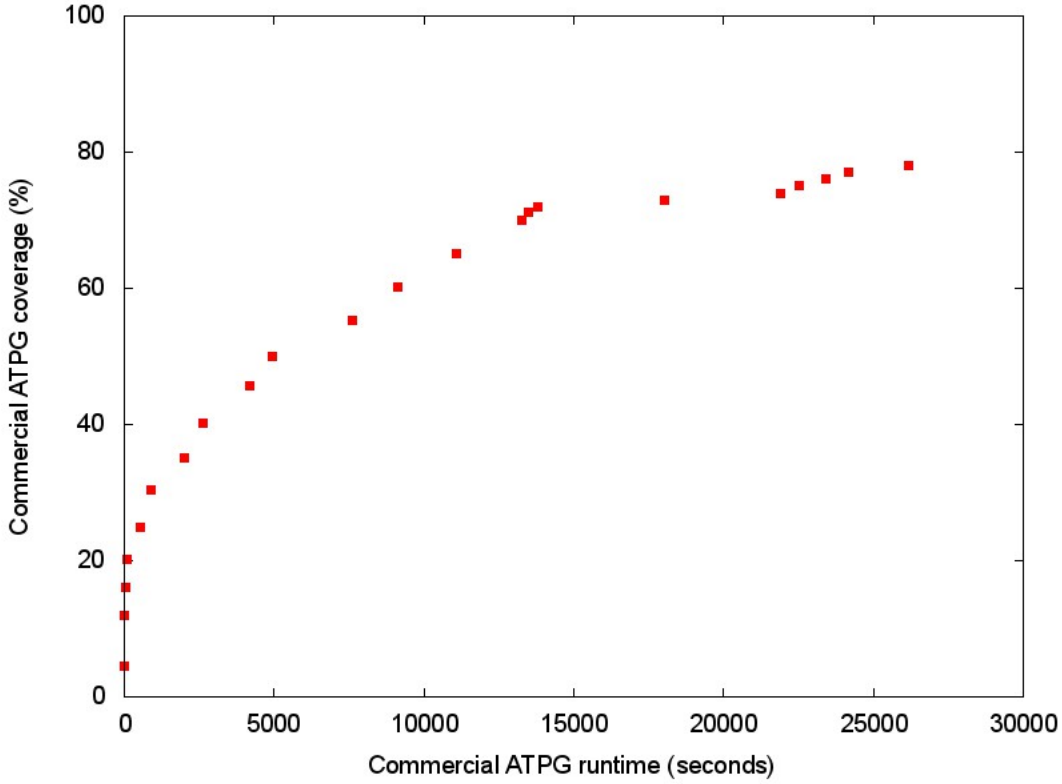

Figure 4.7: Commercial ATPG tool results on OR1200 processor

a commercial ATPG tool was used to generate tests (using sequential ATPG) for faults. Though the ALU was not part of the fault coverage numbers, it was part of the design involved in the test generation process. The results of the commercial ATPG run is shown in Figure 4.7. The graph plots the coverage achieved by the commercial ATPG tool over time.

From the graph we can see that the commercial ATPG tool quickly detects a lot of faults early in the process, but the rate at which faults are detected decreases with time. This suggests that the remaining undetected faults

at any given time are harder to detect. The coverage due to the ATPG tool slowed down considerably at approximately 78%. We stopped the commercial ATPG run at this point and the remaining faults (collapsed undetected faults) were considered as hard to detect faults, and were the focus of our test generation method. If the commercial ATPG tool is given sufficient time it will be able to detect all the faults that we detect using our approach. However, the objective of this commercial ATPG tool run was to extract hard to detect faults.

The ATPG tool results cannot be compared to results from our method for two reasons:

1. ATPG tools do not allow users to specify constraints, so the run had all its inputs unconstrained. Therefore, the tests generated were non-functional and the ATPG technique is not comparable to our approach.

2. ATPG tools perform a fault simulation to drop faults that are detected by existing tests. We did not do a fault simulation, since our focus was on test generation.

Table 4.1 shows the results of commercial ATPG for every module of OR1200 processor. The last column gives the number of collapsed undetected faults which was handled by our test generator.

Table 4.1: Commercial ATPG Coverage Results for OR1200 processor

Module	**Total Faults**	**Undetected Faults**	**Fault coverage (%)**	**# of Collapsed Undetected Faults**
if	3160	621	80.35	328
ctrl	5510	2027	63.21	832
oprmuxes	4226	1113	73.66	378
sprs	9232	961	89.59	393
freeze	170	29	82.94	17
rf	77110	16508	78.59	7444
except	10368	2831	72.69	1263
Overall	109776	24090	78.05	10655

4.5.3 Fault Conditions

The fault conditions presented in Section 4.3.1 were computed using Binary Decision Diagrams (BDDs) and then converted into Sum Of Products (SOPs). To get a compressed SOP representation Espresso (a Boolean minimizer) was applied. Both the BDD package and Espresso were available in the tool ABC [1]. The advantage of the SOP representation is that it does not introduce new variables, i.e., the Boolean function can simply be represented in terms of its input values. However, for some faults, the SOP representation of the fault condition was very large with more than 300 terms. This led to implementation issues because of large files representing the fault injection conditions. So we considered only those faults which had less than 75 SOP terms in their fault condition. In hindsight, a more compact representation like And Inverter Graphs (AIGs) would be more efficient for capturing the fault

conditions. The faults that were dropped due to large SOPs are accounted for in the final coverage numbers of our method.

There was some pre-processing time involved for each fault to compute the fault activation condition, generate the faulty RTL, structurally reduce the faulty RTL based on methods presented in Section 4.4.3 and generate the final SMT BMC formulation. However, since this pre-processing time is low and the test generation time is dominated by the time required to solve the BMC formulation, we only consider the solver run time in our comparisons.

Table 4.2 shows the SOP of the fault condition corresponding to a s-a-1 fault at pin *or1200_if/U44/B* in the OR1200 instruction fetch module. The fault had a single flop, *or1200_cpu.or1200_if.addr_saved[0]* in its combinational fan-out cone. The label of the columns give the names of the inputs in the combinational fan-in cone. The "-" in the table corresponds to a "don't care".

Table 4.2: Example of SOP for fault condition in OR1200 processor fetch module

icpu_ack_i	*saved*	*addr_saved[0]*	*flushpipe*	*if_freeze*
-	1	0	0	1
0	-	0	0	1

4.5.4 SAT-based ATPG

We compared our method against SAT-based ATPG since it is the only other method that is similar to our method in that it targets specific faults. We implemented a version of TEGUS [64] for sequential test generation by unrolling the design across time frames. MiniSAT [7] was used as the

backend solver. We also include run times for the BMC formulation presented in Section 4.3.2 (henceforth referred to as **Naive Observability Method**) to get a relative measure of improvement in test generation time due to the structural observability property presented in Section 4.4 (henceforth referred to as **Structural Observability Method**).

Table 4.3: Running Time and Coverage Results for SAT-based ATPG

Module	**Fault coverage by Commercial ATPG(%)**	**# of Collapsed Undetected Faults**	**SAT-based ATPG**		
			FC(%)	**# TO**	**Avg. Time(s)**
if	80.35	328	84.11	310	96.18
ctrl	63.21	832	65.97	817	83.12
oprmuxes	73.66	378	76.09	354	95.49
sprs	89.59	393	90.85	381	93.71
freeze	82.94	17	99.14	2	64.41
rf	78.59	7444	80.50	7268	97.57
except	72.69	1263	73.48	1209	98.63
Overall	78.05	10655	79.17	10343	96.23

For the BMC runs we used a bound of 6, which is the number of pipeline stages plus one additional time frame. With this bound there are enough cycles for an instruction to take effect even in the presence of pipeline forwarding. A time out period of 100 seconds was chosen and any fault for which a test was not produced within the time out period was classified as undetected. We used EBMC [4] as the RT-level model checker to create the SMT formulation and Boolector [3] as the backend SMT solver.

Table 4.3 shows the fault coverage and run times for OR1200 processor by SAT-based ATPG. The module names are given in the first column followed by the fault coverage by commercial ATPG in the second column. The third column gives the collapsed list of undetected faults. The last row gives the consolidated numbers for the entire design. Column labeled **FC%** is the fault coverage percentage. Column labeled **# TO** is the number of faults that timed out. Column labeled **Time(sec)** gives the average time taken to generate a test for a fault in seconds. This statistic accounts for time due to timed out faults as well. From Table 4.3 we can see that SAT-based sequential ATPG timed out on most faults. Hence it did very little to improve the fault coverage. This high number of time outs was mainly due to the multiplier being included in the fan-out cone of influence of the fault and hence the SAT formula having an instance of the multiplier. SAT solvers choke on solving multiplier instances.

4.5.5 The Naive Observability Method

From Table 4.4 the naive observability method performs better than SAT-based ATPG; however, the test generation times are still very high. The fault coverage percentage is significantly better than SAT-based sequential ATPG. Clearly SMT outperforms SAT in solving these instances.

The graph in Figure 4.8 plots the running times of 832 faults in the control module of OR1200 processor for SAT-based ATPG vs. the naive observability method. This gives an idea of the distribution of the run times. SAT-based ATPG times out on most of the faults; however, the naive ob-

Table 4.4: Running Time and Coverage Results for the naive observability method

Module	**Fault coverage by Commercial ATPG(%)**	**# of Collapsed Undetected Faults**	**Naive Observability Method**		
			FC(%)	**# TO**	**Avg. Time(s)**
if	80.35	328	88.49	161	95.13
ctrl	63.21	832	97.15	59	69.72
oprmuxes	73.66	378	98.26	6	57.46
sprs	89.59	393	93.78	57	90.27
freeze	82.94	17	100	0	43.51
rf	78.59	7444	90.21	463	69.83
except	72.69	1263	92.79	128	96.19
Overall	78.05	10655	93.86	874	76.11

servability approach is able to generate tests for most of these faults. This can be attributed to the better handling of bit-vector data-path operators like multipliers and adders by the SMT solver.

4.5.6 The Structural Observability Method

The Structural Observability Method has by far the best numbers, as seen in Table 4.5. It has almost a 3x improvement over the naive method in addition to improving upon the test coverage numbers. This shows that solvers based on RTL will provide the highest scalability compared to Boolean level solvers. Furthermore, use of structural design constraints can substantially reduce the search space.

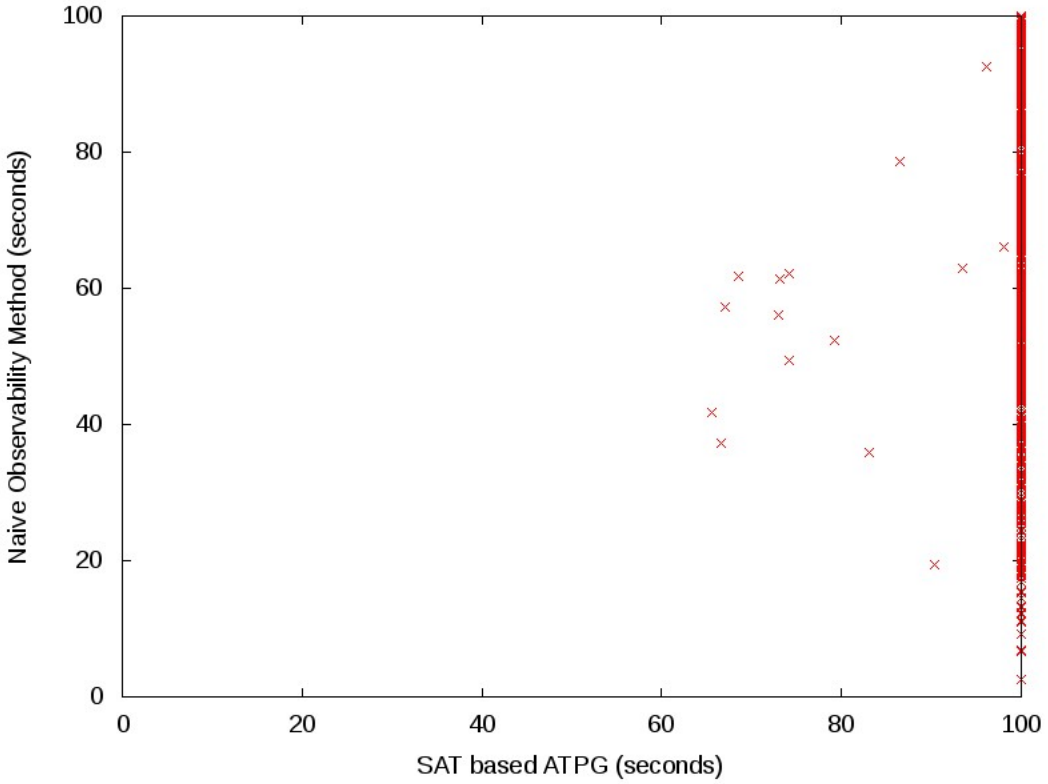

Figure 4.8: SAT-based ATPG vs. the naive observability method run times for OR1200 ctrl module

From a manual study of the OR1200 processor RTL and gate-level netlist, it was not clear if the 286 undetected faults were untestable. We randomly picked out some of the undetected faults and tried to generate tests using structural observability method with unbounded time and memory. Even after running for about 24 hours and consuming significant run time memory, the solver did not return with a result. So these faults are good candidates to be functionally untestable faults.

A reduction in overall running time could have been achieved by per-

Table 4.5: Running Time and Coverage Results for structural observability method

Module	**Fault coverage by Commercial ATPG(%)**	**# of Collapsed Undetected Faults**	**Structural Observability Method**		
			FC(%)	**# TO**	**Avg. Time(s)**
if	80.35	328	98.17	25	23.14
ctrl	63.21	832	99.21	8	21.16
oprmuxes	73.66	378	100	0	19.33
sprs	89.59	393	97.53	12	18.39
freeze	82.94	17	100	0	10.48
rf	78.59	7444	98.37	172	22.85
except	72.69	1263	97.63	69	38.14
Overall	78.05	10655	98.87	286	24.23

forming fault simulation after a test is generated for a given fault. This is because a single test would be capable of detecting multiple faults and those detected faults could have been dropped from the fault list. However, since we were interested in identifying the average run times and making a comparative study we did not filter out the faults detected by each test.

The graph in Figure 4.9 plots the running times of 832 faults in the control module of OR1200 processor for SAT-based ATPG vs. the structural observability method. SAT-based ATPG times out on most of the faults, but the structural observability method is able to generate tests for most of these faults in reasonably time. This can be attributed to three improvements: (1) better handling of bit-vector data-path operators by the SMT solver, (2) struc-

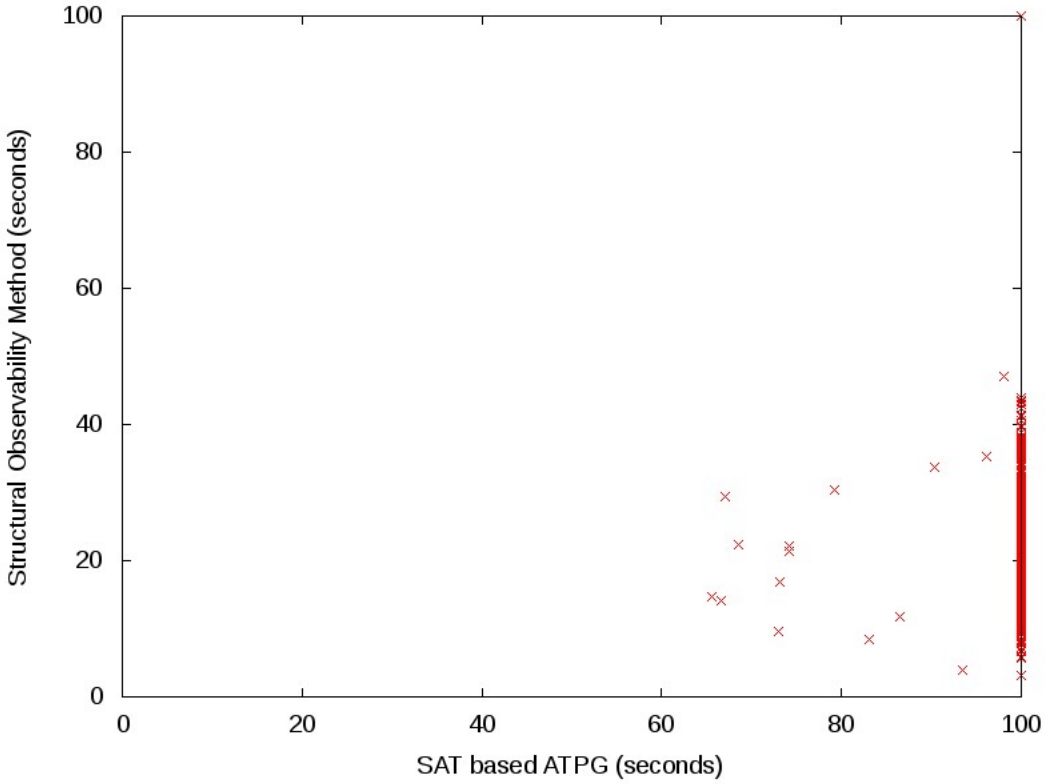

Figure 4.9: SAT-based ATPG vs. the structural observability method run times for OR1200 ctrl module

tural reduction and (3) constraining of the search space by the observability property.

The graph in Figure 4.10 plots the running times of the same 832 faults in the control module of OR1200 processor as earlier for the naive observability method vs. the structural observability method. The structural observability method is able to generate tests for most of these faults in much less time. Some of the faults that timed out for the naive observability approach are also detected. The improvements are mainly due to structural reduction and the

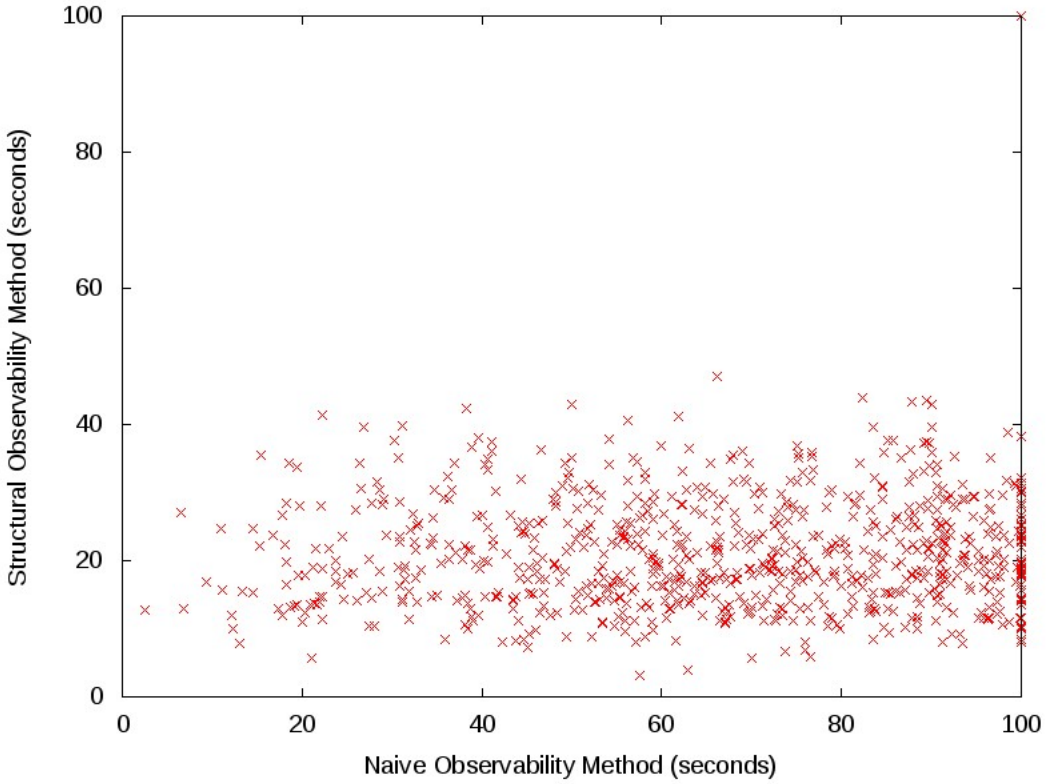

Figure 4.10: The naive observability method vs. the structural observability method run times for OR1200 ctrl module

observability constraints. In the graph we can see that some of the faults that were quickly detected by the naive observability method need more test generation time for structural observability method. This is due to the overhead introduced by the observability property itself. The observability property adds additional variables and logic which stands out for some of the faults that were quickly detected by the naive observability method.

Table 4.6 gives us a few sample tests that were generated by the structural observability method. The first column gives the module in which the

fault exists. The second column gives the gate level pin of the fault. The third column gives the type of stuck-at fault and the last column gives the instruction level test. The length of the instruction sequence varies from 2 to 6, (6 being the BMC bound is the maximum size of the tes)t. The size of the test varies depending on the cone of influence of the fault. Faults deep in the pipeline typically have larger cones of influence and hence required longer tests.

Table 4.6: Sample tests generated by structural observability method

Module	**Pin**	**Fault type**	**Test**
or1200_wbmux	/U10/Y	s-a-0	l.movhi r2, r20, 0x321 l.sb 0x02(r0),r2 l.addi r7, r2, 0x453 l.nop l.ror r8,r21,r7
or1200_if	/U51/B	s-a-1	l.andi r1, r0, 0x83fa l.or r12, r0, r1

4.6 Summary

In this research we have shown a new methodology for generating functional tests for gate level faults using RTL. The improvement in fault coverage and a comparison of the run times of the different methods is summarized in Figure 4.11 and Figure 4.12 respectively. Our methodology is general enough to be applied to any design and can be used as an alternative to sequential ATPG. Though our method is expensive in that it generates a single test for a given stuck-at fault, it is ideal for plugging coverage holes in existing func-

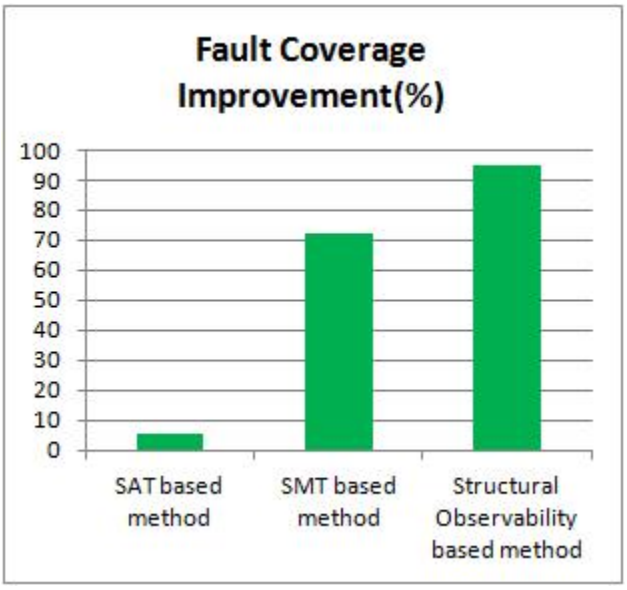

Figure 4.11: Improvement in fault coverage due to different methods for OR1200 processor

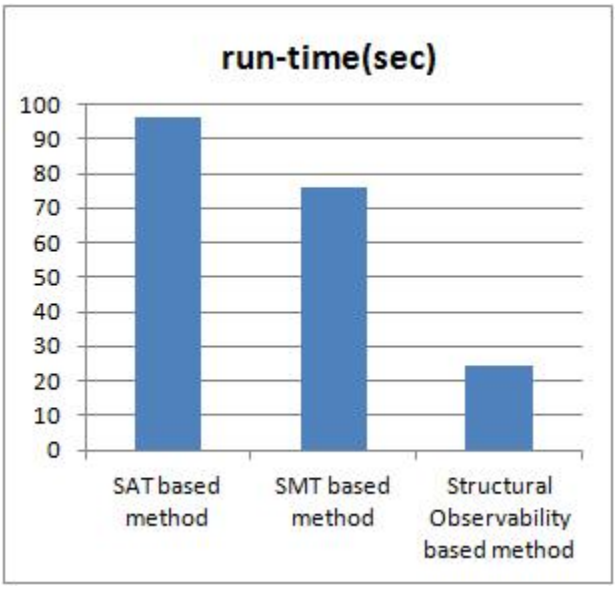

Figure 4.12: Average run time of different methods for OR1200 processor

tional tests by targeting the hard to detect faults and improving the overall coverage. Our experimental results show that solvers that work at higher levels of abstraction perform better than Boolean level solvers and hence scalability can be achieved only by exploiting abstractions at the RT level.

In this work we exploit the advancements made in SMT solver technology. We formulate the BMC problem as a bit-vector formula and rely on a fast bit-vector solver. Further scaling of our method can be achieved by using other abstraction techniques like Array abstractions and under approximation techniques like bit-width abstractions and data-path operator abstractions. SMT solvers also have incremental and unsatisfiable core extraction capabilities which can further be applied for scaling. Our formulation of the test generation problem as a RTL Model Checking problem also makes it possible to exploit advances in High Level Model Checking such as predicate abstraction, which can be applied only to the RT level of abstraction.

Chapter 5

Application of Under-approximation Techniques to Test Generation

In this chapter we present our work which improves the scaling of our test generation methodology introduced in the previous chapter. In order to generate a test for a fault all the behaviors of the DUT are not required. We use this observation to remove some of the behaviors that are irrelevant to the fault under test. This reduces the search space for the model checker, making it possible to scale our original methodology to larger designs. So given a DUT and a stuck-at fault we create an abstraction, called under-approximation, that has fewer behaviors compared to the un-abstracted DUT. A test generated by the model checker on the under-approximate abstraction is also valid on the un-abstracted DUT. We explore two kinds of under-approximations: bit-width reduction and data-path operator approximation.

In bit-width reduction we restrict the domain of bit-vector variables in the RTL. Such a restriction would constrain the search space for the solver, improving running time. If the abstraction is unsatisfiable we use the unsat core to determine if a refinement of the abstraction is necessary i.e., if we need to increase the width of the bit-vector. Such a restriction is an under-

approximation because the variable size is restricted which in turn reduces the number of possible behavors of that variable.

In data-path operator approximation technique all the data-path operators are partially interpreted. Specific operators are chosen to be approximated and a fixed set of rewrite rules are used to replace the original operators with their approximations. Both the range and the domain of these functions are reduced. Such an approximation is an under-approximation because only a subset of the true functionality of those operators is captured.

The contributions of this chapter are as follows.

1. We introduce a highly scalable test generation technique for gate level faults using RTL bounded model checking.

2. We show the effectiveness of under-approximation as an abstraction technique for test generation.

3. We leverage structural insights from the DUT and the fault under test to make under-approximation more effective for test generation.

5.1 Introduction

Given the faulty product state machine $M \times M^f$ we create a structural under-approximate abstraction

$$\overline{M \times M^f} = (S_0, PI, PO \cup PO^f, R \cup R^f, RF \cup RF^f, \overline{T \cup T^f})$$

where, $\overline{M \times M^f}$ represents the under-approximation of the faulty product state machine, S_0 is the initial state constraint, PI is the set of primary inputs, PO is the set of primary outputs in the good machine, PO^f is the set of primary outputs in the faulty machine, R is the set of internal registers in the good machine, R^f is the set of internal registers in the faulty machine, RF is the register file in the good machine, RF^f is the register file in the faulty machine, T is the set of transition relationships between one register state to another in the good machine, T^f is the set of transition relationships between one register state to another in the faulty machine and $\overline{T \cup T^f} \subset T \cup T^f$ i.e., only a subset of the possible transitions is available in $\overline{M \times M^f}$.

Theorem 1. A witness to the observability property O_{fault} in the state machine $\overline{M \times M^f}$ is also a witness to O_{fault} in the state machine $M \times M^f$.

Proof. For simplicity and without loss of generality let us represent the product machine $M \times M^f$ by the Kripke structure $K = (S, I, R, L)$, where S represents the set of states, $I \subset S$ is the set of initial states, $R \subset (S, S)$ is the set of transitions, and $L : S \rightarrow 2^{AP}$ is a labeling of state properties where AP is a set of atomic propositions. Similarly let the under-approximate abstraction $\overline{M \times M^f}$ be represented by the Kripke structure $\overline{K} = (\overline{S}, \overline{I}, \overline{R}, L)$, where $\overline{S} \subset S$ is the set of states, $\overline{I} \subset I$ is the set of initial states, $\overline{R} \subset R$ is the set of transitions.

A witness to O_{fault} in $\overline{M \times M^f}$ can be represented by a path $w = (s_0, ..., s_n)$ through $\overline{K}$ that satisfies O_{fault}, where $\{s_0, ..., s_n\} \in \overline{S}$ and $(s_i, s_{i+1}) \in$

$\overline{R}$ where $i = 0, ..., n-1$. Now since $\overline{S} \subset S$, $\{s_0, ..., s_n\}$ also $\in S$ and since $\overline{R} \subset R$, (s_i, s_{i+1}) also $\in \overline{R}$. Hence path w also exists in $M \times M^f$ and since w satisfies O_{fault}, w is also a witness to O_{fault} in $M \times M^f$. □

By Theorem 1, every test created for a fault on the under-approximation is also a test for the same fault in the DUT. The added advantage is that the under-approximation has fewer states and transitions and hence the SMT solver has to search through a smaller search space. Hence we focus on creating under-approximate abstractions that can be solved easily by the SMT solver but at the same time has enough behaviors to activate the fault and propagate the fault to the output. If a test is not created on the under-approximation, i.e., the solver returns an *unsat* result, then more analysis is required to check if the fault is untestable or if under-approximation needs to be refined. An under-approximation that needs to be refined has too few behaviors to detect the fault.

5.2 Under-approximation Techniques

5.2.1 Bit-width Reduction

In this abstraction we restrict the domain of bit-vector variables in the RTL. These restrictions would imply that each of the bit-vectors would take a smaller subset of the possible values. This reduction in the domain of possible values of the bit-vector may speed up the search. If the abstraction is unsatisfiable we use the unsat core to determine if a refinement of the abstraction

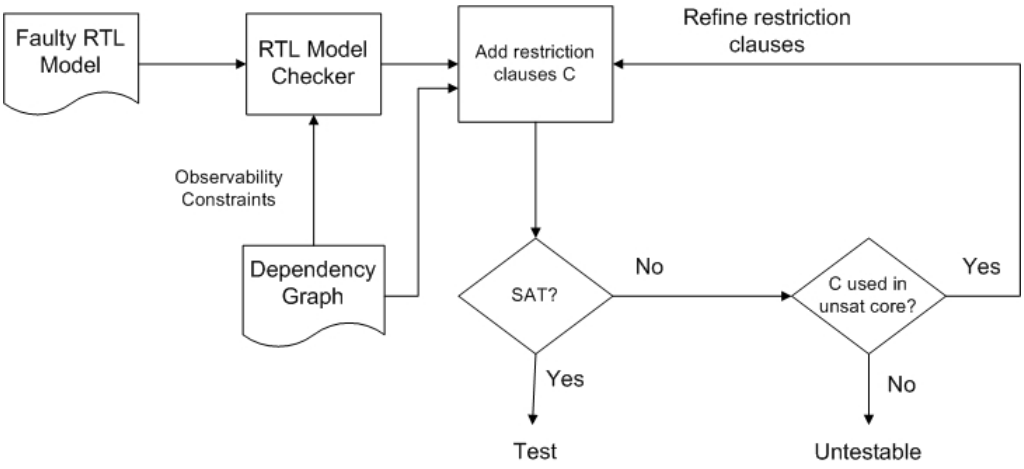

Figure 5.1: Bit-width Refinement loop

is necessary. This loop of abstraction and refinement is shown in Figure 5.1.

In bit-width reduction the most significant bits of the bit-vector are constrained and the rest of the least significant bits are left unconstrained [18]. If w is the size of the bit-vector and the lower n bits are unconstrained then n becomes the effective bit-width. The upper $(w - n)$ bits can be sign-extended, zero-extended or one-extended [18]. Such a restriction is an under-approximation because the variable size is restricted which in turn reduces the number of possible behavors of that variable.

5.2.1.1 Bit-width Encoding

We enforce bit-width reduction by adding additional constraints on the bits of a bit-vector. Let us assume we want to constraint a w size bit-vector

variable v to an effective bit-width of n and sign-extend the higher bits.

$$\bigwedge_{i \in \{n,\ldots,w-1\}} ((v_{n-1} \vee \overline{v_i} \vee \overline{e_v}) \wedge ((\overline{v_{n-1}} \vee v_i \vee \overline{e_v}))$$

Assuming variable e_v to be true enables the constraint and it is disabled otherwise. This is used during refinement if we want to disable a constraint. The above constraints ensures that bit v_i is equal to bit v_{i-1} i.e., the sign bit is extended to the higher bits.

5.2.1.2 Motivating Example

Consider the verilog code shown below, where $in1$, $in2$ and *out* are 32-bit variables.

```
always @(posedge clk)
  case (opcode)
    2'b0: out <= in1 + in2;
    2'b1: out <= in1 - in2;
    2'b2: out <= in1 * in2;
    2'b3: out <= in1 << in2;
  endcase
```

Let us assume that we have to propagate the $D/\overline{D}$, which originates at *opcode*, to *out*. In this case we do not need $in1$, $in2$ and *out* to be 32-bit wide. We can reduce them to 1-bit each, and even under these restrictions it is evident that the $D/\overline{D}$ can be propagated to *out*. The dependency graph for the above code is shown in Figure 5.2. From the dependency graph we can infer that if the $D/\overline{D}$ is propagating to a control dependent variable then we

do not need the full bit-width of the control dependent variable. However if the $D/\overline{D}$ is propagating to a data dependent variable then we may need the full bit-width of the data dependent variable.

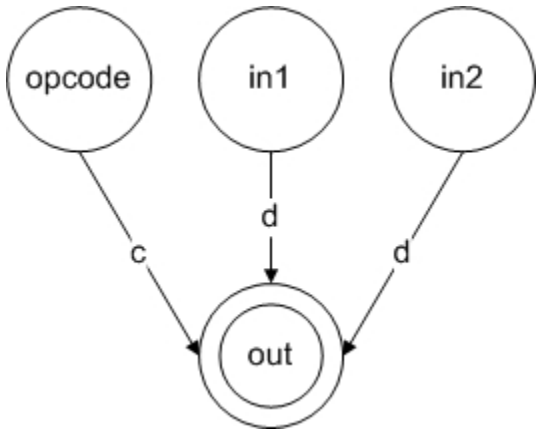

Figure 5.2: Variable dependency graph

It might initially appear that constraining bit-vectors this way might lead to spurious tests. Take the example of a 32 bit counter that computes the next address for the program counter. If we reduce the size of the counter from 32 bits down to say 2 bits then one might think that the counter would count 0, 1, 2, 3, 0 and so on, which would be invalid. But the restrictions only constrains the width of the variable and does not modify the functionality of the counter itself. Hence, the counter would only be allowed to add up to 3 and not beyond that.

5.2.1.3 Refinement Strategies

As shown in Figure 5.1, we initially add restriction clauses to reduce some of the bit-vector variables to get a reduced SMT formula. If the solver returns an *unsat* result we analyze the unsat core to check if SMT formula was truly unsatisfiable. The unsat core gives the variables that were involved in the unsatisfiability. If none of the under-approximated variables appear in the unsat core then no test is possible. If any of the under-approximated variables appear in the unsat core then we refine relax the constraints on those variables. A relaxation of a variable means that we increase the effective bit-width of the variable. This refinement loop allows the solver to maintain the learned clauses across refinements. The width of the good and faulty versions of a signal should always be the same. Hence when we relax the width constraint on a variable, we increase the width of both the faulty and non-faulty version of the signal. This is required from a correctness perspective, otherwise the solver will generate false results. Since the objective is to propagate a Boolean difference to one of the observable points in the design, a spurious Boolean difference might be generated if the width of good and faulty versions of the signal are not maintained to be equal.

A variety of refinement strategies are discussed in [18]. The bit-vector variables to be refined can be identified through the unsat core. A local refinement involves refining only those variables identified in the unsat core. A global refinement involves refining all the variables in the SMT formula. We also have the option of increasing the effective bit-width of the variables by

one at each refinement or double it.

The local strategy is too fine grained and a lot of time is spent in the refinement loop for some faults. The global strategy is too coarse grained and the complexity of the SMT formula increases exponentially at each refinement. Our refinement strategy lies somewhere between local and global refinement. Also we start with the effective bit-width of one for the bit-vector variables and whenever we need to refine some variables we remove all the constraints on those variables. Our initial abstraction is guided slightly by the user. Only the variables affected by the data-path bus and address-bus are reduced. This is because most of the control variables take a specific subset of values. If these variables are reduced then the solver has to refine these variables irrespective of the fault under test. Also the advantages of bit-width reduction comes from reducing the complexity of arithmetic operators which lie in the data-path and address logic.

Our refinement algorithm is discussed in Algorithm 5.1. *dep_graph* is the dependency graph which indicates the dependencies and their types between variables. *unsat_C* is the set of unsat clauses and the function call $refinement\_variables$ (*unsat_C*) returns the set of variables that need to be refined. In this algorithm we start at the variables that need to be refined and then refine all the variables that are directly or indirectly data dependent on them. This is because the $D/\overline{D}$ is possible propagated through these variables.

Consider the verilog example below.

Algorithm 5.1 Forward Refinement of Variable Bit-width

Input: dep_graph, unsat_C
Output: Refined variables

```
 1: RV ← refinement_variables(unsat_C);
 2: push(queue, RV);
 3: while (is_empty(queue)) do
 4:    cur_vertex = front(queue);
 5:    refine(cur_vertex);
 6:    SV = successors(dep_graph, cur_vertex);
 7:    for all succ_vertex ∈ SV do
 8:       if (dep_type(cur_vertex, succ_vertex) == datapath) then
 9:          push(queue, succ_vertex);
10:       end if
11:    end for
12: end while
```

```
reg a, b;

reg [1:0] c, d;

always @(posedge clk)
  c <= {a,b};

always @(posedge clk)
  d <=  c;
```

In this example assume that the $D/\overline{D}$ is in variable a and has to be propagated through c to d. Let us also assume that initially the variables c and d are reduced to 1-bit variables. Since c is reduced the $D/\overline{D}$ cannot be propagated, because a drives the upper bit of c. In the case of local refinement, only bit c will be refined and increased to a 2-bit variable. The variable d will be refined in the next iteration. In the case of graph based refinement, variable d will be refined in the same iteration as variable c, since there is

a data dependency between c and d. In this way, dependency graph based refinement reduces the number of refinement loops.

5.2.2 Data-path Operator Approximation

In this technique all the data-path operators are partially interpreted. This means that hard operators are replaced by an approximation that captures only a subset of the defined functionality. We use fixed rewrite rules to approximate the operators. Both the range and the domain of these functions are reduced. Such an approximation is an under-approximation because only a subset of the true functionality of those operators is captured. The multiplication operator can be rewritten as follows.

- $out = x * y$ is replaced by $(x = 0 \implies out = 0) \wedge (x = 1 \implies out = y) \wedge (y = 0 \implies out = 0) \wedge (y = 1 \implies out = x)$

Similarly, addition can be rewritten as:

- $out = x + y$ is replaced by $(x = 0 \implies out = y) \wedge (y = 0 \implies out = x)$

These rewrite rules completely remove the complexity of addition and multiplication and leave behind functions that are very easy to solve. As long as the $D/\overline{D}$ does not originate at the operator we can rewrite the operator.

5.2.2.1 Motivating Example

Consider the RTL code show below

```
always @(posedge clk)
  case (opcode)
    2'b0: out <= in1 + in2;
    2'b1: out <= in1 - in2;
    2'b2: out <= in1 * in2;
    2'b3: out <= in1 << in2;
  endcase
```

Let us assume that we have to propagate the $D/\overline{D}$, which propagates into *in*, and then to *out*. In this case we do not need the full functionality of any of the operators. The operators can be reduced to propagation functions that simply pass the value at the input to the output and the fault will be detectable under this reduction.

The approximated operators need not always be reduced to propagation functions but any sort of "look up table" can serve as a approximation. The objective is to remove the complexity of performing the computations by the solver. The use of propagation functions, like the ones in the example above, increase the likelihood of propagating the $D/\overline{D}$ towards the observable points.

5.3 Experimental Results

We carried out experiments on an Intel quad-core 2.5 GHz server with 32 GB of RAM. We again used OR1200 processor [8] as the DUT. We again considered faults only in the control part of the circuit since data-path elements like adders and multipliers are easily observable and can be effectively tested. Stuck-at faults were injected in all modules of OR1200 processor netlist except the ALU and then a commercial ATPG tool was used to generate tests (using

sequential ATPG) for faults. Though the ALU was not part of the fault coverage numbers, it was part of the design involved in the test generation process. The coverage due to the ATPG tool saturated at approximately 78%. The remaining faults (collapsed undetected faults) were considered as hard to detect faults and were the focus of our test generation method. Table 4.1 shows the results of commercial ATPG for every module of OR1200 processor. The last column gives the number of collapsed undetected faults which was handled by our test generator.

For the bounded model checker (BMC) [16] runs we used a bound of 6, which is the number of pipeline stages plus one additional time frame. With this bound there are enough cycles for an instruction to take effect even in the presence of pipeline forwarding. A time out period of 100 seconds was chosen and any fault for which a test was not produced within the time out period was classified as undetected. We used EBMC [4] as the RT-level model checker to create the SMT formulation and Boolector [3] as the backend SMT solver.

We implemented the under-approximation techniques as a layer over Boolector.

First we compared the bit-width reduction technique with dependency graph based refinement against local refinement strategy and the results are shown in Table 5.1.

The first column gives the different modules in OR1200 processor. The second column labeled "No abstraction" gives the run time for our original

Table 5.1: Running Time for Bit-width reduction

Module	**Average time(s)**		
	No abstraction	**Local refinement**	**Graph based refinement**
if	23.41	53.03	16.27
ctrl	21.16	38.74	12.49
oprmuxes	19.33	42.61	10.18
sprs	18.39	26.31	8.72
freeze	10.48	32.10	7.15
rf	22.85	34.06	9.17
except	38.14	35.38	17.24
Overall	24.23	35.18	10.62

methodology. The third column and the fourth column give the run times for local refinement strategy and dependency graph based refinement strategy. We can see that dependency graph based refinement is about 2x faster than running the solver without any abstraction. Even though the average runtime of local refinement is higher than running the solver without abstraction, there were cases which were solved in 4x the time taken to solve those instances without abstraction. The reason for this worst case behavior is that a lot of time was spent refining variables along a propagation path. This is where the dependency graph based refinement excels as it avoids the additional refinement iterations by making the refinements more eagerly. Some manual input was involved to decide the initial abstraction. For example, the signal *icpu_dat_i* fetches the next signal from the instruction cache. If this 32-bit signal is reduced then we will not be able to create tests since the opcode is typically determined by the most significant bits. Another bad candidate for reduction is the 5-bit signal *or1200_ctrl.alu_op* which determines the type of ALU op-

eration. Example of a good candidates for abstraction are data-path signals such as *or1200_alu.a*, *or1200_alu.b* and *or1200_alu.result*. Similarly data-path variables in OR1200 register file and write-back muxes are good candidates for reduction. We analyzed the verilog code and determined bit-vectors that should not reduced, i.e., bit-vectors that were going to refined if reduced. The rest of the bit-vector variables were allowed to be reduced.

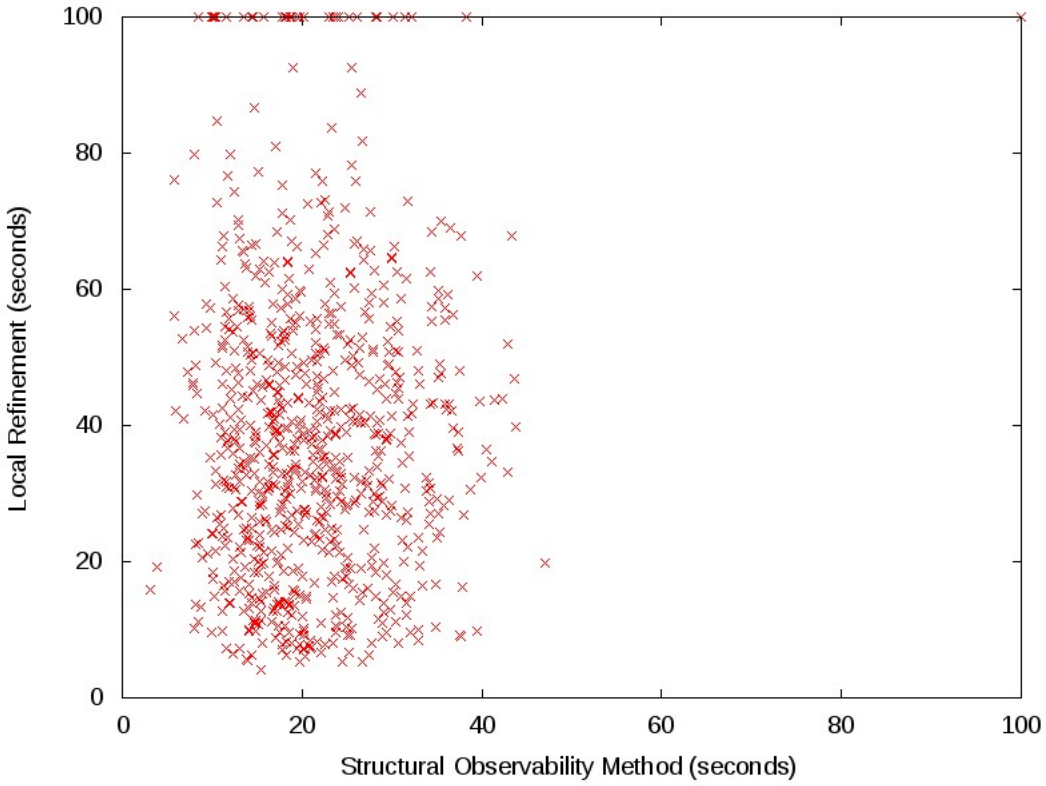

Figure 5.3: The structural observability method (no abstraction) vs. local refinement strategy run times for OR1200 ctrl module

The graph in Figure 5.3 plots the running times for the 832 faults in the control module of OR1200 processor for no abstraction (the structural

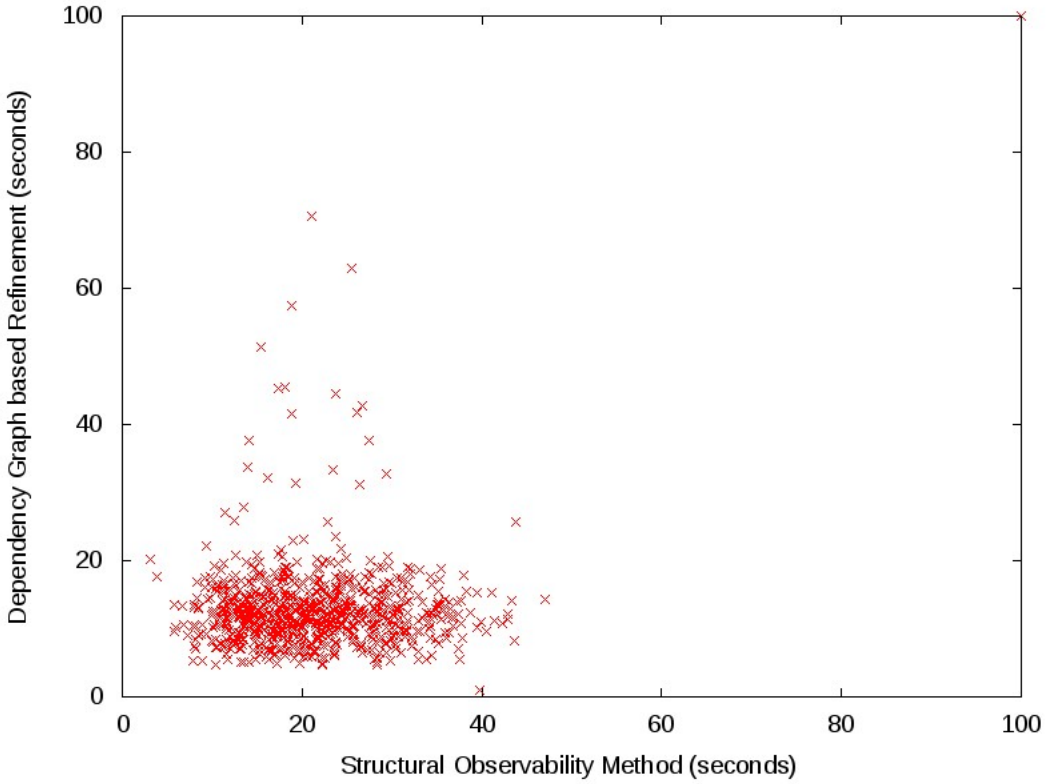

Figure 5.4: The structural observability method (no abstraction) vs. dependency graph based refinement strategy run times for OR1200 ctrl module

observability method) vs. local refinement strategy. The overall run time of local refinement is higher than not using abstraction. As explained earlier, this is because of the overhead of the refinement loop. There are also more faults that time out than when we use no abstraction. However, we can see that there are many instances where local refinement performs much better than when no abstraction is used. This motivated us to develop a better strategy than the local refinement strategy.

The graph in Figure 5.4 plots the running times for the 832 faults in the control module of OR1200 processor for no abstraction vs. the dependency graph based refinement. The overall run time of dependency graph based refinement is lower than when no abstraction is used. There are some faults for which the refinement loop takes more time than when no abstraction is used. The faults that are undetected by dependency graph based refinement but are detected when abstraction is used, are also faults that need more time for refinement. They are detected if the time out period is increased.

Table 5.2: Fault coverage results from structural observability method (no abstraction) for OR1200 processor

Module	**FC(%)**	**# TO**
if	98.17	25
ctrl	99.21	8
oprmuxes	100	0
sprs	97.53	12
freeze	100	0
rf	98.37	172
except	97.63	69
Overall	98.87	286

Table 5.2 gives the fault coverage when using the structural observability method (no abstraction) for the modules of OR1200 processor. Table 5.3 gives the fault coverage with the dependency graph based refinement for the modules of OR1200 processor. Compared to the coverage with no abstraction, the dependency graph based refinement results in a few time outs, resulting in a slight drop in coverage. However, the faults that are undetected by dependency graph based refinement but are detected by no abstraction, can be

Table 5.3: Fault coverage results from dependency graph based refinement for OR1200 processor

Module	**Fault coverage (%)**	**# TO**
if	98.05	35
ctrl	99.21	8
oprmuxes	100	0
sprs	97.53	12
freeze	100	0
rf	98.32	190
except	97.62	72
Overall	98.79	317

detected by dependency graph based refinement if the time out period is increased. The faults that are undetected by no abstraction remain undetected in the dependency graph based refinement. This is because those faults are possibly functionally untestable and having a bit-width reduction based under-approximation will not help in proving the faults untestable. Each refinement of the model takes us closer to the original model that was not abstracted. As mentioned earlier, under-approximations are not suited to proofs.

We carried out similar experiments for operator approximation. We reduced bvmul, bvadd, bvsub, bvshl, bvshr, bvxor and bvor operators in OR1200. Only the operators in the OR1200 alu were reduced. The approximations for each of the operators had to be worked out manually. Most of the approximations were chosen such that one of the inputs was passed on to the output as

is, without any transformation. We used the following rewrite rules:

- $out = x * y$ is replaced by $(x = 0 \implies out = 0) \wedge (x = 1 \implies out = y) \wedge (y = 0 \implies out = 0) \wedge (y = 1 \implies out = x)$
- $out = x + y$ is replaced by $(x = 0 \implies out = y) \wedge (y = 0 \implies out = x)$
- $out = x - y$ is replaced by $(x = 0 \implies out = -y) \wedge (y = 0 \implies out = x)$
- $out = x \ll y$ is replaced by $(x = 0 \implies out = 0) \wedge (y = 0 \implies out = x)$
- $out = x \gg y$ is replaced by $(x = 0 \implies out = 0) \wedge (y = 0 \implies out = x)$
- $out = x \oplus y$ is replaced by $(x = 0 \implies out = y) \wedge (y = 0 \implies out = x)$
- $out = x|y$ is replaced by $(x = 0 \implies out = y) \wedge (y = 0 \implies out = x)$

The run time results are shown in Table 5.4.

Table 5.4: Running Time for Operator Approximation

Module	**Average time(s)**	
	No abstraction	**Operator Approximation**
if	23.41	6.79
ctrl	21.16	8.42
oprmuxes	19.33	7.84
sprs	18.39	8.19
freeze	10.48	5.91
rf	22.85	10.72
except	38.14	12.29
Overall	24.23	8.86

The second column labeled "No abstraction" gives the run time for our original methodology and the third column gives the run time for operator approximation. The run times show that there is a 3x improvement over abstraction-less runs. Operator approximation was applied only to operators on the data-path bus, and operators that are used for address computation like counters were not reduced. This was done mainly because reduction of operators in control logic hampers functionality greatly. Such reductions become counterproductive by producing more *unsat* results. Other than the data-path logic there are no candidates for operator approximation.

The graph in Figure 5.5 plots the running times for the 832 faults in the control module of OR1200 processor for no abstraction vs. data-path operator approximation. We can see that operator approximation improves the run time for all the faults. As explained earlier, the improvement is because all the complexity of the data-path operators is taken out of the SMT formula. Since there is no refinement involved, there is no worsening of run time for some faults as it was in the case of bit-width refinement. The reduction in logic complexity also results in 3 of the faults being proven *unsat*. We will discuss why this cannot be considered as untestable faults when we next discuss fault coverage.

Table 5.5 gives the fault coverage due to data-path operator approximation for the modules of OR1200 processor. We can see that the fault coverage is the same as that of no abstraction. The faults that timed out turned out to be the same faults that timed out with no abstraction. In addition, data-path

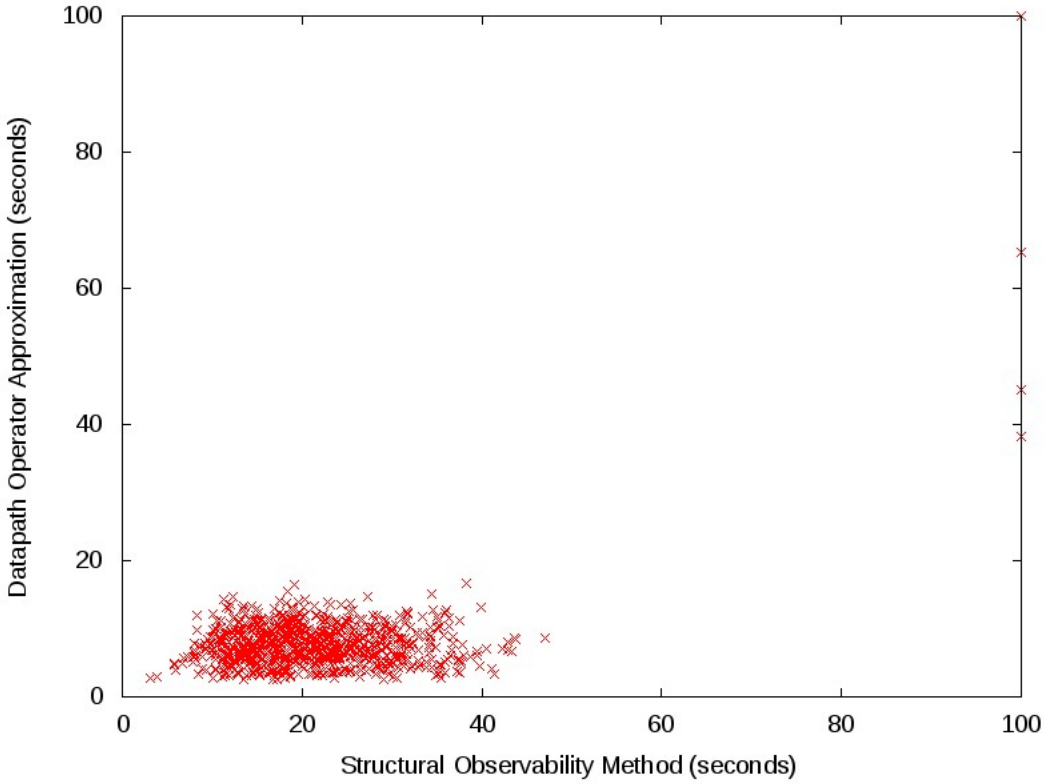

Figure 5.5: The structural observability method (no abstraction) vs. datapath operator approximation run times for OR1200 ctrl module

operator approximation approach was able to return *unsat* result for some faults. These faults timed out and were undetected with no abstraction. We cannot, however, conclude that these faults are functionally untestable since these were unsatisfiable with an under-approximation.

Table 5.5: Fault coverage results for data-path operator approximation

Module	**FC(%)**	**# TO**	**unsat**
if	98.17	20	5
ctrl	99.21	5	3
oprmuxes	100	0	0
sprs	97.53	8	4
freeze	100	0	0
rf	98.37	139	33
except	97.63	48	21
Overall	98.87	220	36

5.4 Summary

Our experimental results show the scalability of the under-approximation technique. We get a 2x-3x improvement in run time over the pure SMT based approach. Bit-vector reduction and operator approximation techniques are very powerful abstractions that are highly suitable to test generation. We leverage the advancements in SMT solvers and we exploit reductions that can be made at higher levels of abstraction to generate tests for gate level faults. Though some manual insights are required for guiding the under-approximations, our method is nonetheless effective for targeting coverage holes in the control logic of large designs. Further scaling of this approach can be achieved by combining bit-width abstraction and data-path operator approximation. Path based search can also be implemented as a layer above these abstractions to achieve even higher scalability. The use of operator approximation suggests that the logic of counter arithmetic with uninterpreted functions [19] can also be used to generate tests for faults in the control logic.

We will explore these approaches in the future.

Chapter 6

Path Based Search for Test Generation

As seen in the previous chapter, creating under-approximations of designs is very useful for speeding up our test generation algorithm. The under-approximations in the previous chapter focused more on behavioral reduction by looking at the semantics of RTL. We also introduced some reductions, namely removing duplicated signals and cone of influence reduction, which were structural in nature. These reductions did not look at semantics but reduced the design by analyzing the RTL structure. We explore another structural reduction, path based search, in this chapter. Instead of solving the entire SMT formula in one run, we solve it piecewise. We try to identify error propagation paths for the fault that are easily solvable for the solver. The paths are chosen using the dependency graph and then we run a bounded model check for the observability property for the chosen paths. Since it will be a smaller SMT formula than the complete SMT formula, the idea is that solver will return an answer faster. If the solver returns *unsat*, then we search through a different set of paths. If the solver returns a counterexample (i.e., a test), then this test will be valid on the original DUT since this kind of reduction is also an under-approximation.

6.1 Introduction

Path based search depends on the SMT solver's ability to solve the SMT formula incrementally. Therefore, the solver needs to able to take in SMT assumptions on the fly. For the search to be efficient the solver should maintain lemmas (learned clauses in the case of SAT) between successive solver calls.

6.1.1 Background: Incremental Satisfiability

We will look at an example of learned clause in the case of SAT. Consider the following CNF clauses that need to be satisfied by a SAT solver.

- $c_1 = (\bar{x_1} \vee x_2)$
- $c_2 = (\bar{x_1} \vee x_3 \vee x_5)$
- $c_3 = (\bar{x_2} \vee x_4)$
- $c_4 = (\bar{x_3} \vee \bar{x_4})$
- $c_5, ..., c_n$

Assume that the clauses c_1, c_2, c_3, c_4 are as above and there are additional unspecified clauses. Let us assume that in the midst of solving clauses c_1, c_2, c_3, c_4 the solver picked up the conflict clause:

$$c_1, c_2, c_3, c_4 \implies (x_5 \vee \bar{x_1})$$

What this means is that $(x_5 \vee \bar{x_1})$ should always be true when clauses c_1, c_2, c_3, c_4 are included in the search. We can see that if $(x_5 \vee \bar{x_1})$ is false, at least one of c_1, c_2, c_3, c_4 will be left unsatisfiable. Such learned clauses will help the search through Boolean constraint propagation in the SAT solver. Discarding these learned clauses will often slow down the search, hence, it is beneficial to keep them. These learned clauses will be added on the fly to the list of CNF clauses that need to be satisfied. A SMT solver also operates in a manner similar to a SAT solver and keeps such a list of learned clauses.

6.2 Path Search Heuristics

Incremental SAT solving was first applied to bounded model checking by Shtrichman [62]. The intuition behind this was that smaller SAT formula, typically have less complexity and are much easily solved. If a counter-example is obtained on a partial SAT formula then it will be valid on the complete formula. We apply this approach to test generation using SMT solvers. The idea that smaller formulae are more easily solved than larger formulae also applies to SMT, but, in addition the complexity is determined by the operators in the SMT formula also.

In our approach, instead of searching through all possible error propagation paths of a fault, we choose specific subset of paths. We use the variable dependency graph to identify which paths need to be explored first. When such propagation paths are chosen the entire fan in cone of the paths have to be included in the SMT formula. This is to ensure that we have sound results.

So this means that when we choose a path to an observable signal in the design (registers primary outputs) we have to include the entire fan in cone of the chosen observable signal. The observability property for this observable signal is synthesized within the chosen fan in cone. This SMT sub-formula will be much smaller than the original formula. If the SMT solver finds a solution to this sub-formula then we have a test. Since the formula was smaller, the solution would be found much faster.

If the SMT solver finds that the sub-formula is unsatisfiable then the error cannot be propagated to the chosen observable signal. It is still possible that fault can be observed at a different observable signal, hence, the search is not complete. We choose a different observable signal and search through all possible paths from the fault to the chosen signal. We do this by adding the additional logic of fan in cone of the newly chosen observable signal to the previous sub-formula. There might be overlap in logic cones, in which case the learned clauses for that part of the logic will already be maintained. The solver will not have to start the search from scratch, hence, a solution might be reached faster. If the learned clauses were not maintained the search would be much more inefficient. We need to search through every observable signal till we find a test.

As an example let us assume that the variable dependency graph shown in Figure 6.1 corresponds to some design. *PO1*, *PO2* and *PO3* are the only observable signals. If we search through path *(v1, v2, v3, PO1)* and find that it is unsatisfiable then we search through path *(v1, v2, v4, PO2)*. If we still

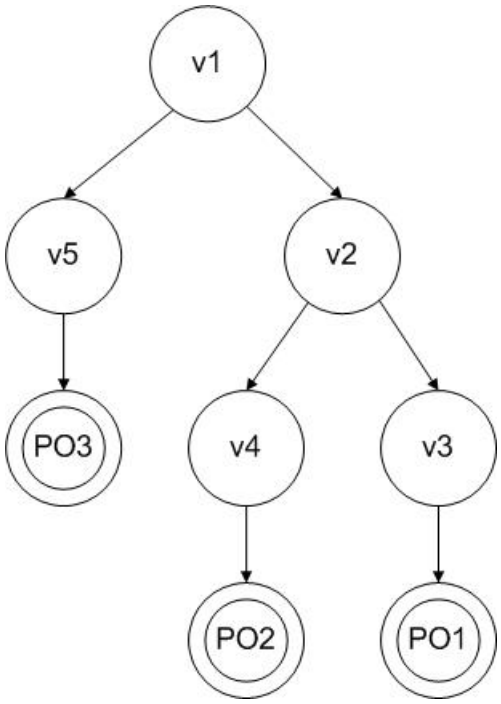

Figure 6.1: Example of path search

do not find a test we search through path *(v1, v5, PO3)* to end the search.

It might initially appear that we could use the unsat core to prune observable signals that might not be reachable. Since the unsat core is the subset of assumptions that are unsatisfiable, any observable signals in the output cone of these assumptions will not be observable. However, most SMT solvers return an approximation of the minimal unsatisfiable core and often the approximations are not very tight. For the search to be sound, the minimal unsatisfiable core should be used. If we use approximations then we might prune out signals that were observable resulting in an incomplete search. Generating the minimal unsatisfiable core is also a NP problem, hence, there are

no fast solutions.

6.2.1 Path Selection

In this section we will look at our path search heuristics in detail. Since there are a lot of observable signals, searching through them one at a time will be inefficient. The number of calls to the solver becomes significant and increases the search time. Hence, we search through sets of observable signals at a time. At each step, the fan in cone logic corresponding to the entire set of observable signal is considered. We also add the observability property corresponding to this reduced cone. Algorithm 6.1 gives the outline of path based search loop.

Algorithm 6.1 Path based search

Input: dep_graph, F_{smt}
Output: sat or unsat

```
1: obs_sig = select_obs_sig(dep_graph);
2: F'_smt = prune_paths_coi(dep_graph, F_smt, obs_sig)
3: result = sat(F'_smt)
4: while ((result == unsat) & exists_obs_sigs(dep_graph)) do
5:    next_obs_sig = select_next_obs_sig(dep_graph)
6:    F'_smt = F'_smt ∪ prune_paths_coi(dep_graph, F_smt, next_obs_sig)
7:    result = sat(F'_smt)
8: end while
9: return result
```

The order in which an observable signal is chosen depends on its distance to the location of the fault. Signals that are at a nearer to the fault are chosen first. There can be a number of paths from the fault location to the observable signal in the variable dependency graph. The length of a given

path is the number of nodes (corresponding to variables) in that path. The depth of a observable signal is the length of the shortest path from the fault location to the observable signal in the dependency graph. We choose this, since shorter paths are more likely to be searched faster, and a fault might be more easily propagated along a shorter path. The second ordering metric we use is the number of data-path dependencies along the shortest path. Observable signals that have more data-path dependencies along the shortest path are chosen first. This is done because error along data-path dependencies are more easily propagated than along control dependencies.

As an example, assume that the verilog code below corresponds to the dependency graph in Figure 6.1.

```
always @(posedge clk)
  v5 <= v1;

always @(posedge clk)
  PO3 <= v5;

always @(posedge clk)
  v2 <= v1;

always @(posedge clk)
  if (a)
    v4 <= v2;
  else
    v3 <= v2;

always @(posedge clk)
    PO2 <= v4;

always @(posedge clk)
```

PO1 <= v3 ;

Now let us assume we need to propagate the $D/\overline{D}$ in *v1* to one of the observable outputs *PO1*, *PO2*, *PO3*. We create the following SMT-lib [10] formula for the fault free DUT.

: assumption (or C_1 (iff v5'1 v1'0))

: assumption (or C_2 (iff PO3'2 v5'1))

: assumption (or C_3 (iff v2'1 v1'0))

: assumption (or C_4 (ite a'1 (iff v4'2 v2'1) (iff v3'2 v2'1)))

: assumption (or C_5 (iff PO2'3 v4'2))

: assumption (or C_6 (iff PO1'3 v3'2))

For the SMT variables above the time frame of the unrolled variable is specified after the tick. We have only shown the SMT formula for the fault free machine, the SMT formula for the faulty machine and the observability property is created as described in Chapter 4. The variables starting with *C_* are the control variables used to determine if we want to include the SMT assumption in the current search or not. We start the search along the path *(v1, v5, PO3)* since it is the path with the least number of variables involved. We set the variables *C_1* and *C_2* to true and the rest of the control variables to false. Let us assume, for the sake of argument, that the path *(v1, v5, PO3)* is unsatisfiable. Then to search along *(v1, v2, v4, PO2)* we set *C_3*, *C_4* and *C_5* to true and the rest of the variables to false. In this way, the control variables help us to determine which SMT assumptions to include in the current search.

6.3 Experimental Results

We carried out experiments on an Intel octa-core 2.9 GHz server with 384 GB of RAM. We again used OR1200 processor [8] as the DUT. Data-path elements were excluded and only faults in control logic were considered. The ALU was part of the DUT model, but no faults were injected into it. Commercial ATPG was used to weed out the more easily detectable faults. The collapsed list of undetected faults from ATPG were classified as hard to detect faults and was the base list of faults for our test generation. Table 4.1 gives the result of the commercial ATPG run on OR1200.

We used a bound of 6 (pipeline depth + 1) for the bounded model checker. A bound of 6 gives sufficient cycles for the error to propagate to one of the outputs. When a bound of 5 was used, most of the faults timed out or were unsatisfiable. For test generation, we used a time out of 100 seconds based on experiments from the previous chapters. Timed out faults were classified as undetected.

Microsoft Z3 [6] was used as the SMT solver. The path based search algorithm was built as a plug-in module using the Z3 API interface. Table 6.1 gives the experimental results for structural observability method. The number of faults that timed out and the average running time for each OR1200 module is listed. Z3 was used as the backend solver.

The experimental results for path based search are given in Table 6.2 We see that the number of time outs have gone up marginally. However, the

Table 6.1: Experimental results for structural observability method

Module	**# of Collapsed Undetected Faults**	**# TO**	**Avg. Time(s)**
if	328	20	18.65
ctrl	832	12	20.72
oprmuxes	378	0	15.49
sprs	393	15	16.57
freeze	17	0	9.68
rf	7444	196	23.21
except	1263	52	36.81
Overall	10655	295	23.94

run time for test generation has given a speedup of 1.45 over a full search. Thus, we get a run time improvement at the expense of fault coverage. The path based search works well for faults whose effects can be propagated through the shorter paths. For faults that are observable at signals that have longer paths, the overhead of incremental satisfiability starts to take over.

The graph in Figure 6.2 plots the running times of the 832 faults in the control module of OR1200 processor for the structural observability method vs. path based search. The average run time of path based search is better than structural observability method. The faults which take more time for path based search than structural observability correspond to faults that need deeper paths. For the faults that time out with path based search but are detectable with path based search, increasing the time out period will detect these faults.

Table 6.2: Experimental results for path based search method

Module	**# of Collapsed Undetected Faults**	**# TO**	**Avg. Time(s)**
if	328	31	12.61
ctrl	832	25	14.39
oprmuxes	378	3	9.87
sprs	393	18	13.68
freeze	17	0	6.24
rf	7444	247	15.94
except	1263	78	24.41
Overall	10655	402	16.40

Table 6.3 gives the fault coverage due to structural observability method (no abstraction) for the modules of OR1200 processor. Table 6.4 gives the fault coverage due to path based search for the modules of OR1200 processor. The increased number of time outs results in a slight drop in fault coverage.

6.4 Summary

Our experimental results show that path based search is a much more scalable approach than a full search. We get a 1.45x improvement in run time over a direct full search. The trade-off with the approach is increased time-outs which results in slightly reduced fault coverage. Nonetheless it is a powerful approach which can be used as a complimentary approach to existing solving techniques. We take advantage of incremental solving abilities of SMT solvers to direct the search piecewise. This approach is similar to choosing the variable

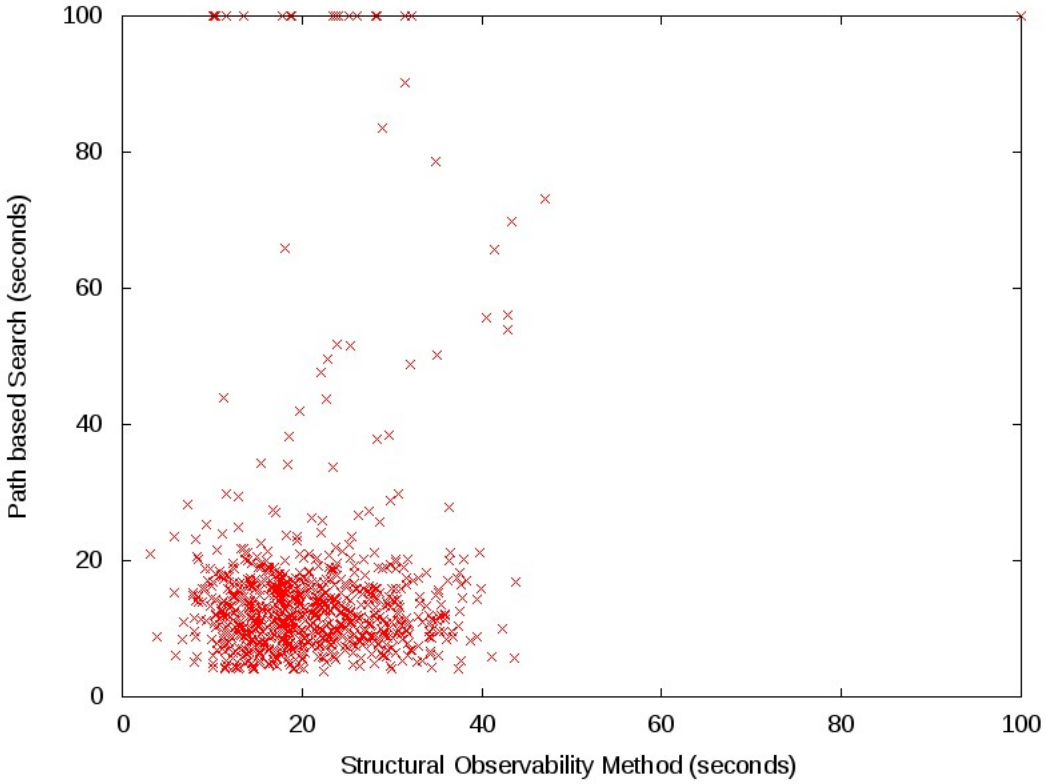

Figure 6.2: The structural observability method (no abstraction) vs. path based search run times for OR1200 ctrl module

ordering in SAT solvers but the granularity in much higher. We choose groups of RTL variables instead of SAT variables. We can further look at combining under-approximation techniques introduced in the previous chapter, since this might give us even higher scalability; we will explore these in the future.

Table 6.3: Fault coverage results due to structural observability method (no abstraction) for OR1200 processor

Module	**FC(%)**	**# TO**
if	98.17	25
ctrl	99.21	8
oprmuxes	100	0
sprs	97.51	15
freeze	100	0
rf	98.28	196
except	96.14	52
Overall	98.87	295

Table 6.4: Fault coverage results from path based search for OR1200 processor

Module	**FC(%)**	**# TO**
if	98.13	31
ctrl	99.03	25
oprmuxes	99.81	3
sprs	97.50	18
freeze	100	0
rf	98.14	247
except	95.59	78
Overall	98.64	402

Chapter 7

Conclusion and Future Work

As we have seen in Chapter 1, the problem of generating at-speed functional tests is very important. With advancements in device fabrication the need for a promising solution is even more pressing. We have presented various scalable techniques for this test generation problem. Our focus was on targeting the stuck-at faults in the a processor.

7.1 RTL Based Test Generation

Our test generation methodology works at the RT-level, making it possible to use RT-level tools. The fault and its corresponding propagation properties are all captured at the RT-level. This in turn makes it possible to use any constraint solver tool in our methodology. Any advancements is high level solvers, such as SMT solvers, can be made use of, as and when they are available.

The drawback of our approach is that the sizes of the propagation properties can get very large quickly. For industrial size processors, the propagation property might be so large that it will incur a significant overhead on its own. Moreover, the cone of influence reduction is useful for only shallow

BMC bounds. As the BMC bound get larger, the shared logic of signals increases and after a point the saving due to this reduction becomes negligible. Hence this approach without any abstractions might not be very suitable for processor with deep pipelines.

7.2 Design Abstractions

We have presented several design abstraction techniques to scale our basic test generation algorithm. Our experimental result shows that a significant speed up is achievable using these techniques. Techniques such as bit-width reduction and data-path operator approximation might involve some manual insight for guiding the abstractions. The sort of information required does not need expert architectural knowledge, but still the methods are not completely automatic. The success of our approach demonstrates the effectiveness of higher levels of abstraction. Use of more abstract logic such as counter arithmetic with uninterpreted functions [19] might provide even higher scalability. These are over-approximation techniques but which have shown to be applicable to out-of-order pipeline verification [41]. Use of uninterpreted functions would be useful especially in generating tests for control path faults.

The other approach of path based search also provides significant speedup. The downside is that the overhead of incremental solving might be too much for some of the faults which might result in more timeouts. This technique can be implemented in combination with the under-approximation techniques which might give us even higher scalability. This needs a very good refine-

ment strategy since the refinement can be structural or on the bit-vector. The refinement overhead for some faults might be very high.

Another interesting direction for future work would be explore use of Craig interpolants [50] for slicing. Craig interpolants can be used to extract fast over-approximate images. From these images we can perform a time based slicing on the unrolled DUT, thereby reducing the size of the model which will be passed on to a solver.

7.3 Other Applications of RTL Fault Injection

For all the presented test generation techniques we used the stuck-at fault model. Our method of mapping stuck-at faults to the RTL model can easily be extend to other applications. For example in the case of delay faults, we can capture fault conditions required to activate the given path using Boolean functions. Once we are able to map the fault to the RTL model the rest of the test generation methodology remains the same. Gurumurthy et. al [34] have used a bounded model checker to target delay defects. However, in their approach all the fault conditions are not captured and hence suffers from the problem of unconstrained gate-level test generation. Since we capture all the fault conditions, this is not an issue in our approach.

We can also apply our approach to generate at-speed functional tests for other problems, such as speed paths. A gate-level timing analysis can be done to identify the paths that need to be sensitized. Then, we can capture all the path sensitization conditions in the same manner as all fault conditions

are captured in this research. Finally, we can use the rest of our methodology for generating functional tests as shown in Figure 7.1.

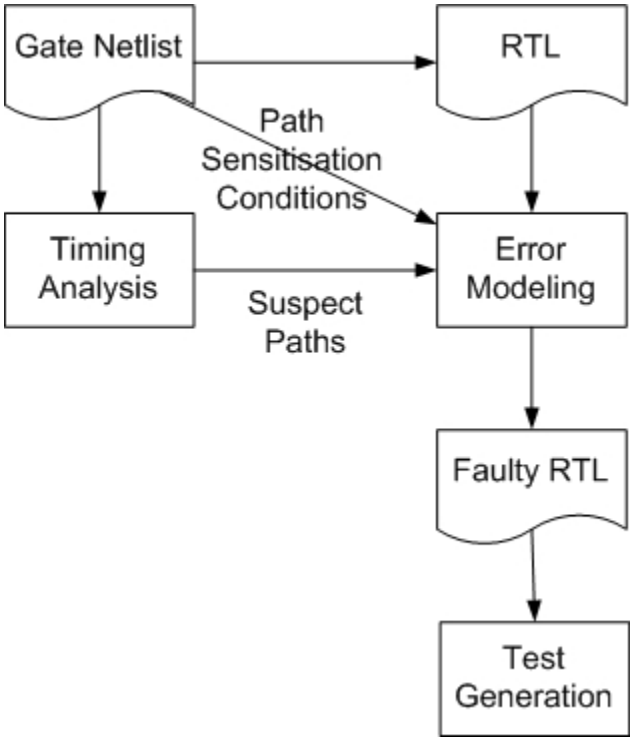

Figure 7.1: Flowchart for generating functional tests for speed paths

Bibliography

[1] ABC. *http://www.eecs.berkeley.edu/ alanmi/abc/.*

[2] ACL2. *http://www.cs.utexas.edu/users/moore/acl2/.*

[3] Boolector. *http://fmv.jku.at/boolector/.*

[4] EBMC. *http://www.cprover.org/ebmc/.*

[5] ITRS report - 2001. *http://www.itrs.net/Links/2001ITRS/Home.htm.*

[6] Microsoft Z3 SMT solver. *http://z3.codeplex.com/.*

[7] MiniSAT. *http://minisat.se/.*

[8] OR1200. *http://www.opencores.org/openrisc,or1200.*

[9] ORPSoC. *http://www.opencores.org/openrisc,orpsocv2.*

[10] SMT-lib. *http://www.smtlib.org/.*

[11] System Verilog Assertions. *www.eda.org/.*

[12] B. Alizadeh and M. Fujita. Guided gate-level ATPG for sequential circuits using a high-level test generation approach. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 425–430. IEEE, 2010.

[13] P. H. Bardell, J. Savir, and W. H. McAnney. *Built-in Test for VLSI*. Wiley, 1987.

[14] S. Bhatia and N. K. Jha. Integration of hierarchical test generation with behavioral synthesis of controller and data path circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(4):608–619, 1998.

[15] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1999.

[16] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying Safety Properties of a PowerPC- Microprocessor Using Symbolic Model Checking without BDDs. In *Proceedings of the Computer Aided Verification*, pages 60–71. Springer, 1999.

[17] D. Brahme and J. A. Abraham. Functional testing of microprocessors. *IEEE Transactions on Computers*, 100(6):475–485, 1984.

[18] R. Brummayer and A. Biere. Effective Bit-Width and Under-Approximation. In *Proceedings of the International Conference on Computer Aided Systems Theory*, pages 304–311. Springer, 2009.

[19] R. Bryant, S. Lahari, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted

functions. In *Proceedings of the Computer Aided Verification Conference*, pages 78–92. Springer, 2002.

[20] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.

[21] H.Y. Chang and G.W. Heimbigner. Lamp: Controllability, observability, and maintenance engineering technique (comet). *Bell System Technical Journal*, 53(8):1505–1534, 1974.

[22] L. Chen, S. Ravi, A. Raghunathan, and S. Dey. A scalable software-based self-test methodology for programmable processors. In *Proceedings of the Design Automation Conference*, pages 548–553. ACM, 2003.

[23] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of programs*.

[24] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero. Fully automatic test program generation for microprocessor cores. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, pages 1006–1011. IEEE, 2003.

[25] V. Dabholkar, S. Chakravarty, I. Pomeranz, and S. Reddy. Techniques for minimizing power dissipation in scan and combinational circuits during test application. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1325–1333, 1998.

[26] E.B. Eichelberger and TW Williams. A logic design structure for LSI testability. In *Proceedings of the Design Automation Conference*, pages 462–468. IEEE, 1977.

[27] R. D. Eldred. Test routines based on symbolic logical statements. *Journal of the ACM*, 6(1):33–37, 1959.

[28] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Transactions on Computers*, 100(12):1137–1144, 1983.

[29] I. Ghosh and M. Fujita. Automatic test pattern generation for functional register-transfer level circuits using assignment decision diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(3):402–415, 2001.

[30] P. Girard. Survey of low-power testing of vlsi circuits. *IEEE Design & test of computers*, 19(3):82–92, 2002.

[31] P. Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. *IEEE Transactions on Computers*, 100(3):215–222, 1981.

[32] S. Gurumurthy, S. Vasudevan, and J.A. Abraham. Automated mapping of pre-computed module-level test sequences to processor instructions. In *Proceedings of the International Test Conference*, pages 10–20. IEEE, 2005.

[33] S. Gurumurthy, S. Vasudevan, and J.A. Abraham. Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor. In *Proceedings of the International Test Conference*, pages 1–9. IEEE, 2006.

[34] S. Gurumurthy, R. Vemu, J.A. Abraham, and D.G. Saab. Automatic generation of instructions to robustly test delay defects in processors. In *Proceedings of the European Test Symposium*, pages 173–178. IEEE, 2007.

[35] G. Hetherington, T. Fryars, N. Tamarapalli, M. Kassab, A. Hassan, and J. Rajski. Logic BIST for large industrial designs: real issues and case studies. In *Proceedings of the International Test Conference*, pages 358–367. IEEE, 1999.

[36] A. Jas, J. Ghosh-Dastidar, and N. A. Touba. Scan vector compression/decompression using statistical coding. In *Proceedings of VLSI Test Symposium*, pages 114–120. IEEE, 1999.

[37] B. Könemann. Built-in logic block observation techniques. In *Proceedings of International Test Conference*, pages 37–41, 1979.

[38] N. Kranitis, A. Paschalis, D. Gizopoulos, and Y. Zorian. Effective software self-test methodology for processor cores. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*.

[39] D. Kroening and S. A. Seshia. Formal verification at higher levels of

abstraction. In *Proceedings of the international conference on Computer-aided design*, pages 572–578. IEEE, 2007.

[40] Daniel Kroening and Ofer Strichman. *Decision procedures*, volume 5. Springer, 2008.

[41] S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in uclid. In *Formal Methods in Computer-Aided Design*, pages 142–159. Springer, 2002.

[42] T. Larrabee. Effecient generation of test patterns using Boolean satisfiability. *Ph.D thesis, Stanford University*, 1990.

[43] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(1):4–15, 1992.

[44] Charles E Leiserson and James B Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1-6):5–35, 1991.

[45] L. Lingappan, S. Ravi, and N.K. Jha. Test generation for non-separable RTL controller-datapath circuits using a satisfiability based approach. In *Proceedings of International Conference on Computer Design*, pages 187–193. IEEE, 2003.

[46] T.H. Lu, C.H. Chen, and K.J. Lee. Effective hybrid test program development for software-based self-testing of pipeline processor cores. *IEEE*

Transactions on Very Large Scale Integration (VLSI) Systems, 19(3):516–520, 2011.

[47] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification*, volume 1. springer, 1992.

[48] P. C. Maxwell, R. C. Aitken, V. Johansen, and I. Chiang. The effect of different test sets on quality level prediction: When is 80% better than 90%? In *Proceeding of International Test Conference*, page 358. IEEE, 1991.

[49] E.J. McCluskey and C-W. Tseng. Stuck-fault tests vs. actual defects. In *Proceedings of the International Test Conference*, pages 336–342. IEEE, 2000.

[50] K. L. McMillan. Applications of craig interpolants in model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–12. Springer, 2005.

[51] G. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 1965.

[52] B. T. Murray and John P. Hayes. Hierarchical test generation using precomputed tests for modules. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(6):594–603, 1990.

[53] P. Parvathala, K. Maneparambil, and W. Lindsay. FRITS - A microprocessor functional BIST method. pages 590–598. IEEE, Proceedings of the International Test Conference, 2002.

[54] A. Pnueli. The temporal logic of programs. In *Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[55] M. Prabhu and J. A. Abraham. Functional Test Generation for Hard to Detect Stuck-At Faults using RTL Model Checking. In *Proceedings of the European Test Conference*, pages 1–6. IEEE, 2012.

[56] M. Prabhu and J. A. Abraham. Application of Under-approximation Techniques to Functional Test Generation Targeting Hard to Detect Stuck-at Faults. In *Proceedings of the International Test Conference*, pages 1–7. IEEE, 2013.

[57] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. S. Reorda. Microprocessor software-based self-testing. *IEEE Design & Test of Computers*, 27(3):4–19, 2010.

[58] J.P. Roth. Diagnosis of Automata Failures: a Calculus and a Method. *IBM journal of Research and Development volume 10*, 10(4):278–291, 1966.

[59] K. Roy and J. A. Abraham. High level test generation using data flow descriptions. In *Proceedings of the conference on European design automation*, pages 480–484. IEEE, 1990.

[60] J. Shen and J. A. Abraham. Native mode functional test generation for processors with applications to self test and design validation. In *Proceedings of the International Test Conference*, pages 990–999. IEEE, 1998.

[61] L. Shen and S. Y. H. Su. A functional testing method for microprocessors. *IEEE Transactions on Computers*, 37(10):1288–1293, 1988.

[62] O. Shtrichman. Pruning techniques for the sat-based bounded model checking problem. In *Correct Hardware Design and Verification Methods*, pages 58–70. Springer, 2001.

[63] A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–511, 1994.

[64] P. Stephan, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1167–1176, 1996.

[65] S. M. Thatte and J. A. Abraham. Test generation for microprocessors. *IEEE Transactions on Computers*, 100(6):429–441, 1980.

[66] R.S. Tupuri, A. Krishnamachary, and J.A. Abraham. Test generation for gigahertz processors using an automatic functional constraint extractor. In *Proceedings of the Design Automation Conference*, pages 647–652. ACM, 1999.

[67] A. J. Van De Goor. Using march tests to test srams. *IEEE Design & Test of Computers*, 10(1):8–14, 1993.

[68] V. M. Vedula, J. A. Abraham, J. Bhadra, and R. Tupuri. A hierarchical test generation approach using program slicing techniques on hardware description languages. *Journal of Electronic Testing*, 19(2):149–160, 2003.

[69] T. J. Verhallen and A.J. Van de Goor. Functional testing of modern microprocessors. In *Proceedings of European Conference on Design Automation*, pages 350–354. IEEE, 1992.

[70] P. Vishakantaiah, J. Abraham, and M. Abadir. Automatic test knowledge extraction from vhdl (atket). In *Proceedings of the Design Automation Conference*, pages 273–278. IEEE, 1992.

[71] L-T. Wang, C-W. Wu, and X. Wen. *VLSI test principles and architectures: design for testability.* Academic Press, 2006.

[72] C.H. Yang and D.L. Dill. Validation with Guided Search of the State Space. In *Proceedings of the Design Automation Conference*, pages 599–604. ACM, 1998.

[73] L. Zhang, I. Ghosh, and M. Hsiao. Efficient sequential ATPG for functional RTL circuits. In *Proceedings of the International Test Conference*, pages 290–290. IEEE, 2003.

Vita

Mahesh Prabhu received his Bachelor's in Computer Engineering from National Institute of Technology Karnataka, India in 2005. He worked for Archpro Design Automation between 2005-2007 as an R&D engineer. He joined the graduate program at University of Texas at Austin in August 2007. He completed got his Masters degree in Computer Engineering in 2009. He has interned at Calypto Design Systems, Obsidian Software, Apple and Intel. Currently he is working as a formal verification engineer at Intel, Austin. His research interests include hardware test generation, formal verification and post-silicon debug.

email: mprabhu@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.