# Reachability Analysis of RTL Circuits Using k-Induction Bounded Model Checking and Test Vector Compaction

Tonmoy Roy

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Masters of Science

in

Computer Engineering

Michael S. Hsiao, Chair

Chao Wang

Haibo Zeng

August 4, 2017

Blacksburg, Virginia

Keywords: RTL Verification, Reachability, k-Induction, Bounded Model Checking, Test Vector Compaction

# Reachability Analysis of RTL Circuits Using k-Induction Bounded Model Checking and Test Vector Compaction

Tonmoy Roy

(ABSTRACT)

In the first half of this thesis, a novel approach for k-induction bounded model checking using signal domain constraints and property partitioning for proving unreachability of branches in Verilog RTL code is presented. To do this, it approach uses program slicing with respect to the variables of the property under test to generate small-sized SMT formulas that describe the change of variable values between consecutive cycles. Variable substitution is then used on these variables to generate the formula for the subsequent cycles without traversing the abstract syntax tree of the entire design. To reduce the approximation on the induction step, an addition of signal domain constraints is proposed. Moreover, we present the technique for splitting up the property in question to get a better model of the system. The later half of the thesis is concerned with presenting a technique for doing sequential vector compaction on test set generated during simulation based ATPG. Starting with a compaction framework for storing metadata and about the test vectors during generation, this work presented to methods for findind the solution of this compaction problem. The first of these two methods generate the optimum solution by converting the problem appropriate for an optimization solver. The latter method utilizes a heuristics based approach for solving the same problem which generates a comparable but sub-optimal solution while having magnitudes better time and computational efficiency.

# Reachability Analysis of RTL Circuits Using k-Induction Bounded Model Checking and Test Vector Compaction

Tonmoy Roy

(GENERAL AUDIENCE ABSTRACT)

Electronic circuits can be described with languages known a hardware description languages like Verilog. The first part of this thesis is concerned about automatically proving if parts of this code is actually useful or reachable when implemented on and actual circuit. The thesis builds up on a method known as bounded model checking which can automatically prove if a property holds or not for a given system. The key insight is obtained from the fact that various memory elements in a circuit is allowed to be only in a certain range of values during the design process. The later half of this thesis is gear towards generating minimum sized inputs values to a circuit required for testing it. This work uses large sized input values to circuits generated by a previously published tool and proposes a way to make them smaller. This can reduce cost immensely for testing circuits in the industry where even the smallest increase in testing time increases cost of development immensely. There are two such approaches presented, one of which gives the optimum result but takes a long time to run for larger circuits, while the other gives comparable but sub-optimal result in a much more time efficient manner.

*To my family*

# Acknowledgments

To make this work complete, I would have to acknowledge the tremendous help provided by various people. First of all I would have to thank Dr. Michael Hsiao for his guidance, ideas and granting me the opportunity to partake in this exciting field of research. His courses on Verification and Testing and Electronic Design Automation helped me immensely.

Moreover I would have to thank Dr. Chao Wang for advising me academically and helping me learn about the field of formal verification which was indispensable for this work. I would also like to thank Dr. Haibo Zeng for graciously agreeing to serve on my thesis committee.

I would have to thank all of my lab-mates at the PROACTIVE lab: Akash Agrawal, Sonal Pinto, Kunal Bansal, Yue Zhan, Aravind V., Tania Khanna and ChungHa Sung. I would especially like to thank Sonal Pinto who I have pestered constantly with questions and ideas which he has engaged with super human patience.

I would also have to thank all of my friends in Blacksburg, who have made my stay here memorable. More importantly, I would like to thank my wife Adrina Halder for her immeasurable love and support which has made it possible for me to achieve my goals.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In the modern world, it can be safe to say that it is impossible to build an appliance or tool without using principles of modern electronics. Almost all types of machines around us use electronics in one form or other. Computers along with the internet that have form the backbone of the modern civilization depend heavily on electronic chips, more specifically on integrated chips (IC). An IC is a semiconductor chip that implements the entirety of an electronic circuit on top of it. Modern VLSI techniques have enabled us to make ICs that can be large enough to implement an entire computer system on a single chip known as a Sytem on a Chip (SoC). At these significantly larger scales, it has become easier for designers to make mistakes. On top of making large scale chip, transistors have become smaller than ever before as well. With the aggressive scaling of transistors, chips are becoming more and more prone to faults. Due to the importance of electronic chips and the modern trend of scaling that has made them prone to errors, now more than ever it has become critical to test and verify them before they are utilized in the intended applications.

## 1.1 Motivation

In the past, integrated circuits used to be small with very simple functionality. It was possible to verify these chips manually. However, keeping pace with Moore's law [1], integrated circuits have become more and more complex. As a result, it has become extremely difficult and exponential effort is spent on design verification. It is claimed [2] that up to 57% of the design effort is spent on verification by chip designers. Consequently it has become essential to use automatic generation of test patterns for chips using ATPG. This has introduced a need for new faster methods of formal and semi-formal design verification techniques making it an active area of research.

A circuit has various levels of design [3]. This starts with the specification, continues through high level description using hardware description language and goes all the way to gate level design. The gate level design is commonly used to model circuit fault and generation of test patterns using ATPG [4]. However, other, higher levels of design abstraction may contain information not present in the gate level design description. For example, the design description using a hardware description language may contain the designer's intention which may not be present in the gate level design. We can consider the cases of signal Don't Care (DC) values, the valid values of states which are present in the Register Transfer Level (RTL) when written using a hardware description language. However all of these information is lost when then design is synthesized into the gate level or the transistor level. The extra information present in the higher levels like the RTL level can be useful for better and faster test pattern generation [5].

Modern electronic chips are more complex than ever before and contain a non-trivially large number of inputs. In most cases it makes it impossible to exhaustively test the chips post fabrication. The exponential relation of the number of vectors to the number of primary

inputs mean that a chip with only 40 primary inputs require $2^{40}$ vectors or more than one billion vectors. Moreover, modern chips are almost always sequential in natures. This means the sequence of the vectors being applied is important as well. As a result, exhaustive search is impossible in a reasonably cost effective time.

To overcome these limitations and generate test pattern for a circuit from the RTL description usually two major types of approaches are taken. Between these two, formal methods as the name suggests, can formally verify a design as well as generate inputs required to trigger a certain type of fault [6]. These methods usually generate very small vector sequence, with which it is possible to quickly verify the design. However, these methods usually do not scale well. Since the chips of today can be very large and complex making formal verification technique take indefeasibly long time to complete, it usually does not make sense to use purely formal verification techniques. Another avenue of test generation is utilization of simulation based technique.

These techniques may use genetic algorithms [7, 8], cultural algorithms [9], or ant colony optimization [5]. Even though these techniques can generate tests with very high coverage very quickly, the generated tests themselves are often very large which can make the verification process more expensive. To aid with the simulation, signal domain based reachability analysis have been proposed to determine the reachability of particular branches in the RTL code.

This work presents a novel approach for improving accuracy of branch unreachability detecting in RTL code by combining k-induction bounded model checking method with previous work of calculating signal domain. Moreover, BMC property partitioning is proposed to handle a special case of unreachability encountered in RTL. This work also introduces a method for vector compacting which utilizes information generated during previously proposed ant colony optimization based ATPG technique.

## 1.2 Contributions of the Thesis

The two main research contributions of this thesis is presented in this sections are presented in this section.

### 1.2.1 Reachability Analysis in RTL Circuits Using k-Induction Bounded Model Checking

In this work, we present a novel approach using an induction-based bounded model checking using a Satisfiability Modulo Theories (SMT) solver for proving the unsatisfiability of the properties. We use program slicing with respect to the variables of the property under test to generate small-sized SMT formulas that describe the change of variable values between consecutive cycles. To reduce the approximation on the induction step, we propose an addition of signal domain constraints.

Moreover, We suggest a technique for breaking up the preceding conditions of a branch and iteratively testing the property to reduce the approximation of the induction step.

Finally, we describe the implementation of a complete tool flow using this method for Verilog designs. We demonstrate the effectiveness of this tool by proving the unreachability of various branches in ITC'99 and the IWLS benchmark circuits which were previously unresolved.

### 1.2.2 Reachability Analysis

In this work, we present a way to formulate the problem of test vector compaction using the sequential test vectors generated during the operation of simulation based ATPG tool. We present a way to formulate the problem of test vector compaction in terms of an optimization

problem that can be converted for solution using an optimization solver.

Moreover, we present a heuristics based method that can solve the same compaction problem with similar results, but in a much more time and computationally efficient way.

We present experimental evidence for these claims by doing modification on the ant colony optimization based ATPG BEACON. We edit the tool by implementing both the optimization based compaction and the heuristics based optimization and test them on Verilog version of ITC'99 benchmarks circuits of various size and complexity.

## 1.3 Thesis Organization

The rest of the thesis is organized as follows:

- Chapter 2 describes the background concepts required to explain the contributions of this work
- Chapter 3 discusses the k-induction based RTL branch reachability analysis using signal domain constraints and property partitioning
- Chapter 4 describes a technique proposed to generate compacted sequential ATPG vectors from BEACON
- Chapter 5 provides concluding remarks for the thesis.

# Chapter 2

# Background

In this section, the details of testing and verification of digital circuits, details of RTL representation of digital circuits, concepts like Satisfiability Modulo Theory (SMT) solver, use of branch distance metric in a fitness function, and tools like Z3 SMT solver, verilator, BEACON etc. are discussed all of which are used for this research and are vital for understanding it.

## 2.1 Testing of Digital Circuits

Digital electronic circuits come in two varieties, combinational and sequential. While combinational circuits generate the outputs based on the current value of the inputs, sequential circuits have memory elements that enables it to calculate the outputs based on the past values of the inputs as well.

Sequential circuits provide more control and are more useful compared to combinational circuits. Since these circuits can save values, they can be used no only to keep past values

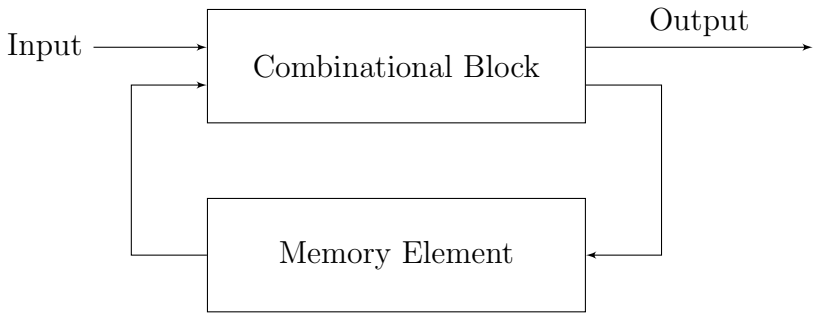



Figure 2.1: Sequential Circuit Block Diagram

of input signals, but also to create constructs such as finite state machines. Any non-trivial circuit these days are sequential circuits. In fact combinational circuits are a subset of sequential circuits.

To understand the working principles of sequential circuits, we can divide it into two parts. These are the combination parts and the memory elements as shown in Figure 2.1. While to test combinational circuits, the ATPG can directly set the values of the primary inputs, it can not do so for sequential circuits. For sequential circuits the ATPG have to figure out what past values to set at the primary inputs such that the memory elements of the current time frame becomes the desired value. This can be stated in a different way. If we assume that the combinational parts of the circuit, without the memory elements is a circuit by itself, then the inputs to the memory elements will be the primary outputs of this circuit (along with its original primary outputs) and the outputs of the memory elements will be the primary outputs of this circuit (along with the original primary inputs). Now the ATPG does not have the ability to affect these new primary inputs or observe the new primary outputs directly either. The only way for the ATPG to set these inputs is to set the actual primary inputs of the circuits to a certain value a few time frames in the past in way that will set the pseudo-primary inputs in the desired way. This concept is known as sequential circuit unrolling and it is shown in detail in Figure 2.2.

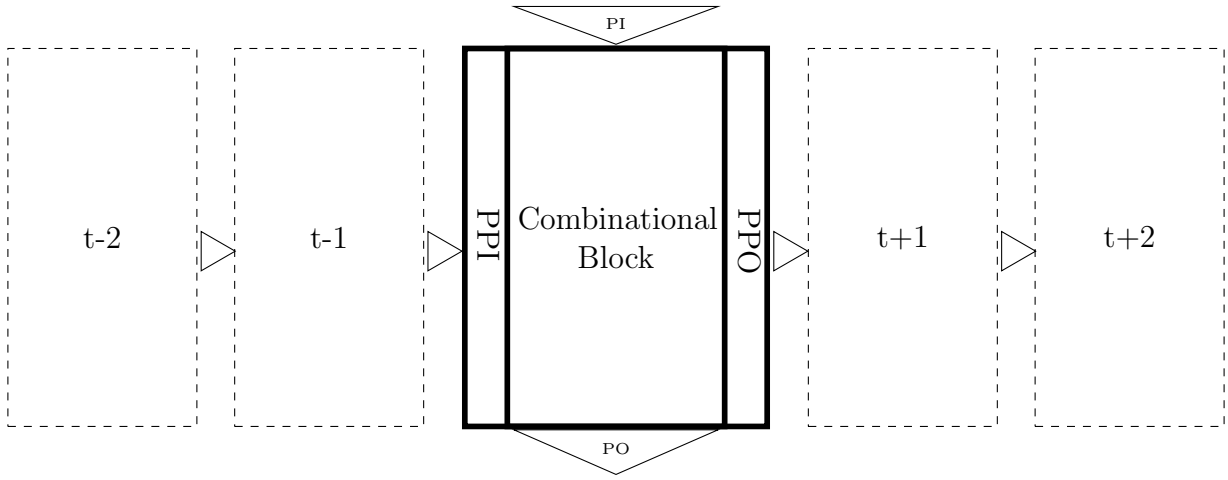



Figure 2.2: Unrolled Sequential Circuit

Purely deterministic formal ATPG methods unroll the circuit for a certain number of times. However unrolling the circuit more than a few number of times cause state space explosion that may become impractical [10]. Other methods use purely simulation based approach, like ant colony optimization algorithms [5, 11] that are more practical. But these methods suffer from lack of guidance. Specifically in cases of finite state machines there may be hard to reach branches that require a specific pattern of vector sequences that these methods may be unable to reach.

More recently, hybrid approaches utilizing both formal methods like symbolic execution and simulation based approaches have proven to be practical, capable of obtaining high coverage, and at the same time able to reach hard to reach branches as well [12, 13, 14]. Most of these approaches however generate very large test sizes which can be a limiting factor for these kind of ATPG algorithms. To reduce the test size a viable option may be to guide the simulation with a branch distance metric that can enable the simulation to reach the hard to reach branches more quickly.

## 2.2 Register Transfer Level Description

Modern electronic circuit design process is too complicated to be done without electronic design automation (EDA). Electronic design automation enables designers to make the design in different layers of abstractions. One of the most common top level of abstraction is the Register Transfer Level. This level of abstraction defines the behavior of the circuit without going into the implementation details [15]. RTL can be designed using Hardware Description Languages (HDL). These languages are human readable and similar to common programming languages. But instead of describing a program to be ran on a processor, these languages describe the behavior of a digital circuit which can be converted into a implementable circuit by EDA tools

Two common hardware description languages are Verilog [16] and VHDL [17]. While both of these languages are widely used, this work is focused on verilog. Verilog has two types of statements, one of them is synthesizable, and the other is non-synthesizeable. Non-synthesizeable code is used mainly to write testbench and aid with the simulation. On the other hand, synthesizeable verilog statements can be implemented into actual circuit. Even though our tools can handle both types of statements, for experimental purposes we only consider synthesizable verilog.

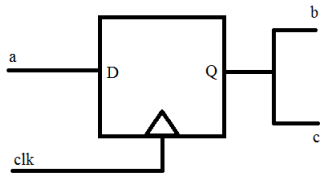
Verilog can be used to describe both sequential and combinational parts of a digital circuit. The sequential parts are usually enclosed in an always block that needs the clock signal as an argument. Inside each sequential block the code for the logic is very similar to any other programming language like Java or C++. However, verilog has different types of assignments. One of them is blocking assignment and the other is non blocking assignment. While blocking assignments trigger immediate assignment to the signal on the left hand side inside a sequential block, the non-blocking assignments take effect only after the entire

```
1 module top (
2 clk, a, c
3 );
4 input clk;
5 input a;
6 output c;
7 reg b;
8 always @ (posedge clk )
9 begin
10   b = a;
11   c = b;
12 end
```

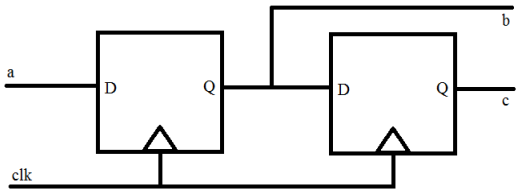
(a) Code Blocking Assignments

```
1 module top (
2 clk,a,c
3 );
4 input clk;
5 input a;
6 output reg c;
7 reg b;
8 always @ (posedge clk )
9 begin
10   b <= a;
11   c <= b;
12 end
13 endmodule
```

(b) Code Non-Blocking Assignments




(c) Blocking Assignment Synthesized




(d) Non Blocking Assignment Synthesized

Figure 2.3: Comparison Between Blocking and Non-Blocking Statements in Verilog

sequential block has completed executing. This nuanced difference needs to be considered when doing static analysis on verilog code. The difference between these two assignments can be seen with the verilog code and the resulting circuit diagram in Figure 2.3

## 2.3 Verilator

Verilator [18] is an open-source, cross-platform compiler that can convert synthesizeable Verilog code into a cycle-accurate C++ or System C code that can be used for simulation. In this work, the converted C++ code was used to automatically generate the verification test vectors. Verilator was designed with speed of simulation in mind to be able to simulate very large scale Verilog design quickly. Figure 2.4 shows an example of a Verilog code and

RTL design in Verilog

```
1  always @ (posedge clock or
       posedge reset)
2  begin
3    if (reset == 1'b1) begin
4      outp <= 1'b0;
5    end
6    else begin
7      case(inp)
8        1'b0: outp <= 1'b1;
9        1'b1: outp <= 1'b0;
10     endcase
11   end
12 end
```

Converted C++ code

```
1  void Vtop::_sequent__TOP__1
2  (Vtop__Syms* __restrict vlSymsp)
       {
3    Vtop* vlTOPp = vlSymsp->TOPp;
4    if (vlTOPp->reset) {
5      ++(vlSymsp->__Vcoverage[0]);
6      vlTOPp->outp = 0;
7    } else {
8      ++(vlSymsp->__Vcoverage[3]);
9      if (vlTOP->inp) {
10       ++(vlSymsp->__Vcoverage
             [2]);
11       vlTOPp->outp = 0;
12     } else {
13       ++(vlSymsp->__Vcoverage
             [1]);
14       vlTOPp->outp = 0;
15     }
16   }
17 }
```

Figure 2.4: Sample Verilog code to C++ conversion using Verilator

the corresponding C++ code generated by Verilator using that Verilog code.

The Verilator-generated C++ code is represented as an object which models the Verilog circuit. The object has various methods to manipulate and simulate the circuit and various properties that can be used to read or write any signal. The *eval* method of this object can be called to evaluate the circuit for the current inputs set. To convert Verilog into executable C++ code, Verilator separates the code into sequential and combinational parts. The combinational parts are evaluated until convergence when the eval function is called. The sequential parts depend on the corresponding clock and reset signal that have to be provided. For this work, we had implemented an interfacing code that would connect with Verilated design and handle all of this.

Verilator utilizes loop unrolling to make sure that the converted code does not contain any loops within a single cycle. Moreover, all function calls are inlined in place making analysis on the generated C++ code easier to handle. Moreover, Verilator cleans up the code to make sure all the branches in the code are represented using only if-else statements. Finally, it has option to enable branch coverage instrumentation, which is described in more detail in Section 2.4.

## 2.4 Branch Coverage

To approximate the quality of the test vectors generated, metrics are needed. The most common types of code coverage are branch coverage and line coverage. A branch is defined as any decision point in the code in the form of if-else, or switch case statements. A branch is considered to be covered if by running the generated test vectors, the predicate of the branch is evaluated true (or false for else) at least once. For synthesizeable RTL code, where loops are unrolled and function calls are inlined in place, 100% branch coverage would ensure 100% line coverage as well. A test that can generate higher branch coverage will be able to identify more faults compared to a test with lower branch coverage. So branch coverage can be used as a metric for the quality of a test vectors generated by ATPG.

In this work, Verilator was used to convert synthesizeable RTL code into cycle-accurate C++ simulation code. As described in Section 2.3, Verilator converts all switch-case statements into equivalent if-else statements. So all the branches are if-else blocks. The converted code can be instrumented by Verilator so that there is a counter present for every branch in the converted code. Thus, when the circuit is simulated, it was easy to count the branches with non-zero value for the counters to get the value of the branch coverage for any given test.

## 2.5 Satisfiability Modulo Theory Solver

Satisfiability Module Theory models decision problems represented by logical formulae given a background theory such as linear integer arithmetic, bitvector operations etc. For example, the problem $x > 1$ and $x < 3$ has one solution in the linear integer arithmetic theory which is $x = 2$. A SMT solver is a tool which given a SMT problem in the form of a SMT formula can determine whether the formula is satisfiable and provide an example solution if it is in fact satisfiable. To solve an SMT problem, the SMT problem needs to be converted into an SMT formula in the form of assertion clauses. If the solver is evoked after providing all the clauses, then the solver will determine whether the problem is satisfiable given the clauses and the background theory. If the problem is satisfiable, then the SMT solver can be asked to provide a model. This model can be seen as one possible solution for the SMT problem that the solver was able to find. SMT solvers have been used in test generation [19].

In the recent years, there have been extensive research done with regards to SMT solvers. Utilizing state-of-the-art algorithms, data structures, heuristics, and optimization of implementation details, the recent SMT solvers have become more robust and faster. An important driving factor behind the exponential improvement of SMT solvers can be attributed to the annual competition for SMT and SAT solvers known as SMT-COMP.

## 2.6 Z3 SMT Solver

Z3 [20] is an open source SMT solver software published by Microsoft Research. Z3 implements various API in various programming languages including Java, C, C++, and Python. In this work we modified verilator to get the signal domain and to generate the branch distance metric. Since verilator is written in C++, it was convenient to utilize the C++ API

provided by Z3.

Since we are concerned with RTL code, we utilized bit-vector variables for all the signals in the design. To use Z3, a context needs to be created where any signal can be declared. Each signal in the RTL code was declared in a single context of Z3. For bit-vectors in Z3, the bit-width of the signals or any constants can be provided during deceleration. Solvers have to be called with a context. All clauses provided to the solver are considered to be in conjecture form by the solver.

## 2.7 BEACON

BEACON is a branch oriented evolutionary ant colony optimization method for automatic test pattern generation of digital electronic circuits. In effect BEACON is a bio-inspired meta-heuristic. It combines an evolutionary search technique and Ant Colony Optimization [5]. This combination enables it to improve the test pattern search capability immensely. To facilitate the search BEACON first uses verilator to convert verilog RTL description into cycle accurate C++ simulation code. This code is instrumented by verilator to get branch hit count and branch coverage. To utilize the converted C++ code, BEACON generates a interface code, which can be compiled together with the verilated C++ code to generate a dynamically loadable shared object library. BEACON finally loads this circuit object with which it is able to simulate the circuit on a cycle by cycle basis. The pre-processing step required for BEACON is shown in Figure 2.5

After completing the pre-processing step, BEACON uses ant colony optimization techniques to generate tests to achieve high branch coverage on the given circuit. The process initializes with a fixed number of ants who walk randomly over the circuit. After the initial walk is finished, the pheromone map on the branches are updated by reinforcement and evaporation.

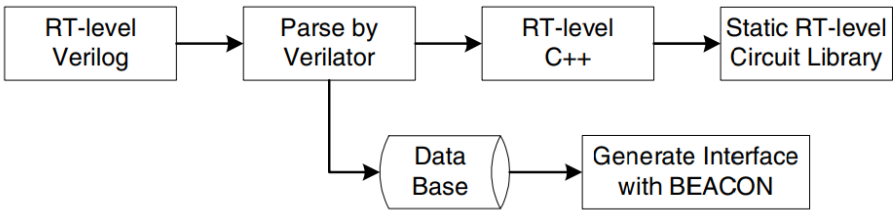



Figure 2.5: Preprocessing Step for BEACON[5]

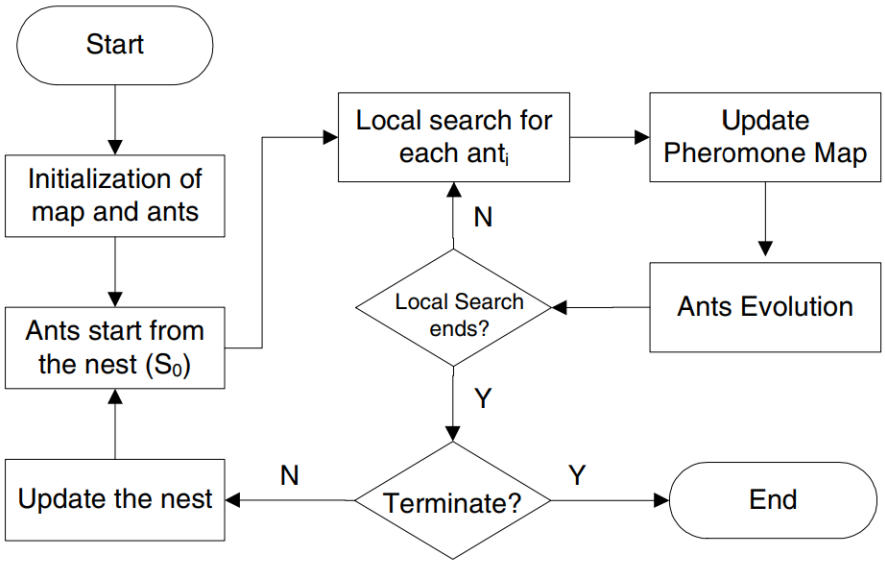



Figure 2.6: Block Diagram of BEACON Search[5]

After which the ants are evolved according to their fitness values and until they can not achieve any new branch coverage for a certain number of clock cycles. At which point, the starting state is updated with one from the newly discovered states and the whole process is repeated again. The guidance framework for BEACON is shown in Figure 2.6.

## 2.8 Control Flow Graph and Data Flow Graph

Many compiler level optimization and static analysis tools utilize the concept of Control Flow Graphs (CFG) proposed by Allen [21]. The CFG is represented using a graph $G(V, e)$.

The basic blocks of code are represented as the vertices, $V$, and the flow of executing between these basic blocks is represented by the edges, $e$. A basic block in a computer program is a number of program statements that meet the following conditions:

1. Each block can only be entered via the first statement.

2. Each block may only contain one exit statement that leads to another basic block.

3. All statements must execute sequentially within a block.

To form the CFG the final statements of each block is taken as execution targets and are used to create the graph edges. Many core analysis of a program like loop optimization and identifying unreachable program segments can be done utilizing this transformation and formation of the CFG.

Operations which rely upon the data assigned to an operand during a prior operation is known as data dependency. Control dependencies on the other hand are the dominance frontier of a node in the reverse CFG. This represents the necessary control path required to active a specific operation. A Data Dependency Graph(DDG) can be generate by conducting data flow analysis on the CFG of a program. Dependencies between register assignments, control statements and previously assigned data can be elaborated using DDG. Data dependency analysis can be used to do program scheduling, software change impact analysis and compiler analysis [22, 23, 24]. In case of RTL code, the data dependency is not only from one variable to another, it is across clock cycles as well. For blocking statements, a variable depends on the value of another variable from the previous cycle. And for non-blocking statements, this dependency is on the preceding statements (which can be from a previous clock cycle as well).

## 2.9 Static Single Assignments

The concept of Static Single Assignment (SSA) [25, 26] come from compiler design research where it is used for code simplification and optimization. At a basic level, the concept of SSA states that every variable should be assigned only once in a program. To handle single variable assigned multiple times, on real programs, versions of the same variable is created and used multiple times. Once a value is assigned to a particular version of a variable, every time the variable is needed, this particular version is used. And subsequent updates to this variable is handled by creating a brand new version of the variable and using that until any further subsequent update. For example, in the following code:

---

$x \leftarrow y$

$y \leftarrow x + 1$

$y \leftarrow y + 1$

$x \leftarrow y$

---

After applying SSA, the assignments would be updated as

---

$x \leftarrow y$

$y_1 \leftarrow x + 1$

$y_2 \leftarrow y_1 + 1$

$x_1 \leftarrow y_2$

---

In this example, '$y_1$' and '$y_2$' are versions of the same variable 'y'. As it can be seen, every time the value of 'y' is updated using an assignment, a new version of the variable is created. In the first statement y is used without any update, so the original version of that variable is used to update 'x'. On the second statement, 'y' is updated, so a new version of it is created. The same thing happens again on the third statement as well. The variable 'y' is finally used in the last statement to update the variable 'x'. For this reason, a new version

of 'x' is created again.

The flow of variable values become clearer after application of static single assignment to the code. This also allows the conversion of assignment statements into equality constraints which can be given directly to the SMT solver while keeping the model assumed by the code flow.

## 2.10 Program Slicing

The method where a program is decomposed automatically by analyzing its data and control flow is known as program slicing [27]. To extract the program slice of a program with respect to a specified subset of the program's behavior, it is required to reduce the program to a minimal form which still produces that specified behavior. This reduced independent program known as a "slice" should ensure that it faithfully represents the original program within the domain of the specified behavior.

Program slicing for RTL code for hardware would differ from program slicing in software domain, in that a statement may not impact the specified behavior in the current cycle, but it still can impact the behavior on any of the subsequent cycles.

## 2.11 Bounded Model Checking

In traditional bounded model checking, a Boolean formula is constructed that is satisfied if and only if the system under consideration can realize a finite sequence of state transitions to reach the state of interest [28]. For this, a finite length k is selected. A symbolic search is conducted for this given length k. Since this search is symbolic, this ensures all path

segments of length k are searched simultaneously.

Bounded model checking is very popular for proving safety and liveness properties in digital circuits. However, BMC have been applied on RTL code as well [29] with moderate success. However, the underlying assumption with BMC is that it can only prove the satisfiability or unsatisfiability of a property within a finite length $k$ starting from the initial state.

While Bounded Model Checking can be very effective for test generation and verification of a circuit, it can become a much more powerful tool by including other constraints to reduce the search space [30]. With the added constraints BMC can conduct the search within a smaller search space to be able to find the goal in magnitudes of time faster and sometimes even be able to find goals that were previously hard to reach.

## 2.12 Bounded Model Checking with k-Induction

To circumvent the problems with the basic BMC presented in the previous section, a feasible alternative is to prove that a formula is k-inductive [31, 32]. The k-induction method has been successfully applied to verify finite state machines using a SAT solver. Moreover, the k-induction method has been applied on software verification as well [33].

k-induction bounded model checking consists of two steps. These are the base case and the induction step. Let $I(s)$ encode the set of initial states and let $T(s, s')$ denote the encoding for the transition relation from state $s$ to state $s$. Again, let $\phi(s)$ denote the states that satisfy the assertion condition. Then the base case can be written as: $I(s_0) \wedge T(s_0, s_1) \wedge ... \wedge T(s_{k-1}, s_k) \wedge (\phi(s_1) \vee ... \vee \phi(s_k))$ and the induction step can written as: $\neg\phi(s_0) \wedge T(s_0, s_1) \wedge ... \wedge \neg\phi(s_k) \wedge T(s_{k-1}, s_k) \wedge \phi(s_{k+1})$

Intuitively this can be thought of as follows. In the base case, we try to prove that the

assertion condition $\phi$ cannot be reached in $k$ cycles. After that in the induction cycles we try to prove that if the assertion condition $\phi$ does not hold for $k$ cycles, that it cannot be satisfied on the $k+1$-the cycle either.

It is typical to start with $k = 1$ and increase $k$ by 1 for every iteration [34]. However, since our tool is concerned with branches that cannot be reached easily by other simulation of formal method based test generation techniques, or cannot be proven by simpler signal domain based unreachability testing, we increased $k$ by 5 instead.

## 2.13 Signal Domain Based Reachability Analysis

Signal domain based analysis for prediction of branch reachability in RTL code is presented in [35]. The process is as follows. First, the RTL code is instrumented for branch coverage, where each basic block is an unique identifier. Each basic block is then used to extract assignments that are present in those blocks and are labeled with corresponding block identifier. These assignments give an over approximation for the relevant signals' domains. It utilizes all the the activating conditions and all the preceding conditions as well as all the assignments for all variables in these conditions. It generates an assignment table to reason about all the possible assignment combinations that can satisfy reaching a certain branch. For each signal in the activating condition, the associated assignment are added to the assignment graph. These assignments form the first level of the graph. Next, each assignment added to the graph is checked for references to other signals. Any new signals found in those references are added as potential predecessor assignments to the first level assignment, effectively creating a new graph level. This is continued until a fixed point is reached or the depth level limit is reached. This graph creates a representation of the possible assignments relevant to a target activating condition. Finally each path in this graph is used to generate

an unique SMT constraint. This SMT constraint can be given to a SMT solver and if the solver returns unsatisfiable for all possible assignment combinations then the branch in question can be guaranteed to be no reachable. This was a list of branches that are certainly not reachable can be produced. It must be mentioned that, the reachability can only be proved only if the assignment is acyclic, i.e., there is no assignment where the signal updates its own value.

# Chapter 3

# Reachability Analysis in RTL Circuits Using k-Induction Bounded Model Checking

## 3.1 Introduction

The modern day computing is heavily dependent on digital electronic circuits, composed of large, complex system of chips (SOCs). Safety critical systems, such as medical equipments, brakes and control in cars and planes have started to use these SOCs extensively for the immense flexibility they can provide. Thus, it has become paramount to be able to verify the operations of these large and complex chips before deployment to a safety critical operation.

Modern SOCs are increasing in complexity. Even the simpler ones can utilize multiple core processors with multiple peripherals like counter-timers, real-time timers, etc., and multiple external interfaces like USB, Ethernet, USART, SPI, etc. With so many parts

working together, it becomes extremely difficult to test and verify them. It is estimated that verification can consume as much as 57% of the design time [2], making innovation to aid verification much more urgent.

There has been plenty of research done for the generation of test stimuli for digital chips. These mainly fall in two categories. On one hand, there are approaches that focus on simulation based test generation. These methods can include cultural algorithms [9], evolutionary algorithms [7, 8, 36, 37] and ant colony optimization techniques [5]. On the other hand, there have been some methods proposed that combine formal and stochastic (deterministic techniques with simulation) methods to generate test stimuli. [11, 19]. All of these approaches generally succeed at generating test stimuli for majority of the cases under consideration. However, trying to trigger corner cases have been noted to be extremely difficult. Furthermore, for some hard corners, proving that the corners are impossible to be reached can be just as difficult. This can cause the test generator and formal approaches to waste much effort trying to find test stimuli for branches which may in fact be completely unreachable.

Signal domain based analysis on RTL code has been proposed that can prove the unreachability of code blocks without unrolling [35]. However, due to the aggressive abstraction of the approach, the signal domain based techniques may be limited in scope as modern SOC design are complex enough to have branches that require signals from multiple state machines that need to be triggered in a particular sequence to reach.

To solve the problem of unreachability in Verilog RTL code, we propose a novel approach for induction-based bounded model checking using techniques from program slicing by leveraging signal domain constraints. Initially, our algorithm utilizes Verilator [18] to cross compile the given Verilog code into cycle accurate functionally equivalent C++ code. During the conversion, the abstract syntax tree (AST) of the C++ is extracted and parsed to generate a representation of the C++ code. The reachability of a certain code block is converted into

a formula in disjunctive normal form (DNF) using this AST. All the variables that are used in the formula along with all other variables that these variables depend on are extracted from a Data Dependency Graph(DDG) of the code. The set of all these variables are used to generate the program slice for each variable. This program slice formally describes the transition between cycles for each of the variables in question. All of these formulas are converted to a cycle-bounded SMT formula for solving with an SMT solver. This by itself is similar to a bounded model checking which is capable of proving the unreachability of a branch for a given cycle bound. Z3 SMT solver [20] is called twice using this BMC formula, first for the base case, and later with signal domain constraints for the induction step. We further increase the accuracy of the results when needed by splitting up the safety property and iteratively checking each part for the special cases where applicable.

The main contributions of our work presented in this paper is as follows:

- We describe a technique for performing k-induction bounded model checking on Verilog RTL code using extracted signal domain constraints. We describe the formation of a property using the activating and the preceding conditions of a branch that satisfies the reachability of the branch. Furthermore we convert the RTL code using program slicing with regards to variables in this property.
- We suggest a technique for breaking up the preceding conditions of a branch and iteratively testing the property to reduce the approximation of the induction step.
- We describe an implementation of the described approach and present the experiments run on various benchmark circuits using this implementation. We believe our implementation is able to prove the unreachability of various branches in these benchmark circuits that were previously unresolved.

The rest of this paper is organized as follows. Section 3.2 describes the motivation behind the

presented work. Section 3.3 presents the methodology in detail. The experimental results are discussed in Section 3.4. Finally, Section 3.5 concludes the chapter.

## 3.2 Motivation and Related Works

In this section we describe how hard to reach or unreachable are introduced into digital design.

The main reason for a branch to become hard to reach is if it requires a specific sequence of inputs. Since most modern designs have numerous inputs and can have thousands of registers, it may be impossible to find the particular sequence of inputs required to reach the branch in a practical time. As such simulation based methods may not be suited for reaching these branches. Deterministic methods are better suited to generate these values. However, naive implementation of these branches can be computationally too expensive to be practical. These issues are worsened when the branches are deeply nested as is common for large circuits.

On the other hand, branches in a design become unreachable if there is bug in the design preventing certain combination of register values to be every possible, which are required for the said branch. Again, sometimes branches are designed to handle unexpected register values which never occur under normal operating conditions. These register values will never occur during normal circuit operations. These kind of branches are unreachable for a test generation apparatus or during simulation as they consider normal operating conditions only. Sometimes discovery of unreachablity can also point to other potential design mistakes which may not have been discovered otherwise.

To handle hard to reach and unreachable branches, simulation based methods [5, 7, 8, 9, 37]

let the input vector generation process run for a set number of iterations. If the branch under scrutiny is not reached in this time, then it is considered to be unreachable. This however results in wasted efforts.

Purely symbolic methods like STAR [38] or hybrid method like HYBRO [19] can fail because of path explosion problem. Moreover, often, these methods can only guarantee a property like unreachability for a certain number of cycles starting from the initial state and cannot guarantee anything outside the range of unrolling of the circuit.

## 3.3 Methodology

The high level procedure for our approach is as follows. First, the Verilog RTL code is converted to cycle accurate C++ simulation code using Verilator. In the process, Verilator generates a file with the AST for the C++ code as an artifact. Our tool reads and parses this AST file to generate an internal representation of the C++ program. The tool also requires a target branch to prove unreachability of. The activating condition and the preceding conditions of this branch is combined to formulate the safety property for the bounded model checking. After that, the tool iterates through the AST to construct the CFG and DDG to get a list of variables that the branch depends on. This list of variables is used to generate a SMT formula of the program slice of the code. After that induction based bounded model checking is executed on the SMT formula to try to find a path to satisfy the original safety property. All of these steps are described in detail in the following subsections.

### 3.3.1 SMT formula generation

To be able to perform bounded model checking on a system, a formula describing the transition between states of the system is required. In case of a typical finite state machine, this formula consists of variables encoding the states and the inputs to the FSM. However, for RTL code that has not been explicitly converted to a FSM, this formula would consist of all register variables and the inputs those registers depend on. Our tool traverses through the AST starting from the root of the top module and generates a list of formulas that describe the value of a variable for cycle t+1 with respect to the values of the cycle t. After that the AST is analyzed to extract the CFG of the code and the DDG for the variables used in the activating and preceding conditions of the target branch. These are then used to generate a set of variables required for program slicing. Before the final traversal of the AST, the variables are divided into three categories, variables that do not impact some other value in the current cycle, which include but are not limited to all variables assigned using blocking statements, variables that do impact some other values in the current cycle, and arrays. We do not handle arrays that impact some other value in the current cycle. This division allows the first type of variables to be handled very efficiently using single traversal of the abstract syntax tree. For this, Every node in the AST is used to call the formula generation function recursively. These function calls have the set of the variables in question as argument. Each node returns an expression for the variable in question based on its children nodes if it can in a hashmap, where the variable is the key. For blocking statements, the expression from the last node in the control flow graph for a set of statements is returned. On if statements where only one of the branches assign the variable in question, the other branch is assumed to have the assignment from the next node, or the expression for the variable value from the previous cycle is used. On the other hand, if the variable in question is not impacted by the node itself or by any of the children in the node, then an empty expression is returned. If a

variable does not have any expression returned to the top module, then it is assumed to be equal to the expression for the variable value from the previous cycle. This way the formula for a particular cycle $t$ is generated and then variable substitution is used to get the formula for any $t$ without further needing to traverse the entire AST. The algorithm for this is given in .

For variables that do impact the current cycles, the AST is simply traversed and each variable update is marked with new unique id. Moreover, for arrays the array selection is encoded using *ite* where the index expression of the array is in the condition of the *ite*. For example, for the following Verilog code:

```
1  reg [15:0] A[3:0];
2  input reg [1:0] addr;
3  output reg datao;
4
5  always @ (posedge clock or posedge reset)
6  begin
7    if (A[addr] + A[(addr + 1) & 3] > 33)
8      datao <= 1;
9
10 end
```

Our tool would generate the following SMT expression as the property to test the reachability

of the only branch:

$$ite(addr = 0, A_0, ite(addr = 1, A_1, \\ ite(addr = 2, A_2, A_3))) + ite((addr + 1)\&3 = 0, A_0, \\ ite((addr + 1)\&3 = 0, A_1, ite((addr + 1)\&3 = 0, A_2, A_3))) \\ > 33$$

Here $A_i$ represents $A[i]$ at cycle t, and is encoded in the SMT formula as 'A<i>'.

**Handling Non-blocking Statements**

While blocking statements are more popular than non-blocking statements for Verilog design, non-blocking statements are used in significant amount of designs to be ignored. Our method for SMT formula generation using program slicing can handle blocking statements efficiently. Moreover, even in cases where non-blocking assignments are used, but static single assignments are not required as described in section 2.9, the original formula generation algorithm works perfectly fine. Again, even if the variable does need static single assignments, but if the assignments are never used during the current cycle, they can be ignored as well. To identify these variables, the control flow graph for the current cycle of the program is traversed. The variables that do not fall in any of these categories have an impact on the state of the circuit in the current cycle. These variables cannot be handled by the base algorithm and have to be specially handled using concepts of static single assignments. The identification of these current cycle impacting variables are generated using algorithm using the function 'SSA_INIT' in Algorithm  by traversing the CFG of the code for the current cycle.

After identification of these cycle impacting variables, just like compiler static single as-

**Algorithm** Formula Generation from Program Slicing

```
1: function PGM SLICE(n,V)
2:     if n is NULL then return NULL
3:     else if type(n) is IF then
4:         E_n ← PGM SLICE(next(n),V)
5:         V_c ← vars(E_n) − v
6:         E_1 ← PGM SLICE(if_stmt_head(n),V_c)
7:         E_2 ← PGM SLICE(els_stmt_head(n),V_c)
8:         E_r ← {}
9:         for all v ∈ V_c do
10:            if v ∈ E_1 ∧ v ∉ E_2 then
11:                E_r[v] ← ite(expr(cond(n)), E_1[v], v)
12:            else if v ∉ E_1 ∧ v ∈ E_2 then
13:                E_r[v] ← ite(expr(cond(n)), v, E_2[v])
14:            else if v ∈ E_1 ∧ v ∈ E_2 then
15:                E_r[v] ← ite(expr(cond(n))E_1[v], E_2[v])
16:            end if
17:        end for
18:        return E_r
19:    else if type(n) is ASSIGN then
20:        E_n ← PGM SLICE(next(n),V)
21:        E_r ← {}
22:        if lhs(n) ∈ V ∧ lhs(n) ∉ E_n then
23:            E_r[lhs(n)] ← expr(rhs(n))
24:        end if
25:        return E_r
26:    else
27:        return PGM SLICE(next(n),V)
28:    end if
29: end function
30: V ← relevant non-array non-cycle-impacting variables
31: E ← PGM SLICE(ast_root,V)
32: for all t ∈ {0..k} do
33:    for all (v, e) ∈ E do
34:        ADD(solver,expr(v_{t+1} == e_t))
35:    end for
36: end for
37: ADD(relevant array and cycle-impacting variables)
```

**Algorithm** Formula Generation for Current Cycle Impacting Variables

**Require:** $V$: Set of Current Cycle Impacting Variables

**function** SSA INIT($n$, $U$)
  **if** type(n) is VAR **then**
    $v \leftarrow$ name(n)
    **if** v $\in U$ **then**
      ssa_parent(n) $\leftarrow U[v]$
      ssa_id(v) $\leftarrow max\_ssa\_id(v) + 1$
    **end if**
  **else if** type(n) is IF **then** SSA INIT($cond(n)$,{})
    **if** else(n) is NULL **then**
      else(n) $\leftarrow U$
    **end if**SSA INIT($if(n)$,{}) SSA INIT($else(n)$,{})
    **for all** $u \in$ Variables Modified in children(n) **do**
      $W[u] \leftarrow n$
    **end for**
  **else if** type(n) is ASSIGN **then**
    $W[lhs(n)] = n$
  **end if**
  $W \leftarrow \{W$, SSA INIT($next(n)$,$W$) $\}$
  **return** $W$
**end function**
**for all** $v \in V$ **do**
  **for all** $i \in \{1..max\_ssa\_id(v)\}$ **do**
    $x \leftarrow encode\_ssa(v, i)$
    **if** ssa_parent(n) is NULL **then**
      $m \leftarrow root$
    **else**
      $m \leftarrow ssa\_parent(x)$
    **end if**
    $e \leftarrow expr(x_{t+1} ==$PGM SLICE$(m, \{x\}))$
    ADD($solver$,$e$)
  **end for**
**end for**

signment handling, our tool creates a new version of the variable every time the variable is updated. Any use of this variable is taken from the last generated version of the variable. If the variable was defined inside a if-else block with different values for different branches, then the variable versions are used with the appropriate if-else predicated inside an *ite* expression. If only one of the mutually exclusive branches update the variable, then the non updating branch is deemed to be using the last updated variable version. And if such a version does not exist in any of the preceding statements, then the expression from the previous cycles is used inside the *ite* expression. This way the expressions for all non-blocking assignments are generated as well. One point to consider is that we do not handle all variables at the same time with a set of variables sent as argument to each node as this is not a very common case in most design and the algorithm becomes too complex. The whole approach is detailed in Algorithm .

**Handling Arrays**

Although some SMT solvers support theory of arrays, they are assumed to be large arrays that are assumed to have number of array access much smaller than the size of the array. If this is not the case, and if the array size is constant, then it can be more efficient to encode the array by enclosing it into multiple *ite* blocks where the condition would be an equality made from the selection expression and the index of the value under consideration. For example, if we have an array named $A$ with size 4, and if the array is selected using the variable $i$, then the array selection can be emulated with the following formula

$$ite(i = 0, A[0], ite(i = 1, A[1], ite(i = 2, A[2], A[3])))$$

Our tool handles arrays in a similar way. First, each array indexed variable is split by

**Algorithm** Formula Generation for Array Variables

**Require:** $v$: Variable in consideration, $N$: Size of array

**function** P($n$,$v$,$i$,$p$,$e$)
    **if** n is NULL **then return** $e$
    **else if** type(n) is IF **then**
        $e_c \leftarrow expr(cond(n))$
        $e_1 \leftarrow$ P($if\_stmt\_head(n)$,$v$,$i$,$p \wedge e_c$,$e$)
        $e_2 \leftarrow$ P($els\_stmt\_head(n)$,$v$,$i$,$p \wedge e_c$,$e$)
        $e_r \leftarrow ite(e_c, e_1, e_2)$
        **return** P($next(n)$,$v$,$i$,$p$,$e_r$)
    **else if** type(n) is ASSIGN **then**
        $s \leftarrow expr(array\_sel(lhs(n)))$
        $r \leftarrow expr(rhs(n))$
        $e_r \leftarrow ite(p \wedge (s == i), r, e)$
        **return** P($next(n)$,$v$,$i$,$p$,$e_r$)
    **end if**
**end function**
**for all** $v \in$ Related Array variables **do**
    **for all** $i \in \{0..N\}$ **do**
        $e \leftarrow$ P($n$,$v$,$i$)
        ADD($solver$, $e$)
    **end for**
**end for**

appending the array indexes surrounded by special characters '<' and '>' at the end of the array name. For example the array index $A[0]$ is represented with $A < 0 >$. When array values are used somewhere in the code, it is replaced with a similar compound *ite* statement where $i$ is replaced with the expression of the array selection extracted from the code and $A[x]$ is replaced with $A < x >$ where $x \in \{0, 1, 2, 3\}$. For example, for the following Verilog code:

```
1  reg [15:0] A[3:0];
2  input reg [1:0] addr;
3  output reg datao;
4
5  always @ (posedge clock or posedge reset)
6  begin
7    if (A[addr] + A[(addr + 1) & 3] > 33)
8      datao <= 1;
9
10 end
```

Our tool would generate the following SMT expression as the property to test the reachability of the only branch:

$$ite(addr = 0, A_0, ite(addr = 1, A_1,$$
$$ite(addr = 2, A_2, A_3))) + ite((addr + 1)\&3 = 0, A_0,$$
$$ite((addr + 1)\&3 = 0, A_1, ite((addr + 1)\&3 = 0, A_2, A_3)))$$
$$> 33$$

Here $A_i$ represents $A[i]$ at cycle t, and is encoded in the SMT formula as 'A<i>'.

The expression for each array indexed value for the next cycle is generated using this method as well. For each index $i$ of an array, a program sliced expression is generated using *ite* statements where the condition for the *ite* is an equality with $i$. For example, if we consider the following code:

```
1  reg [15:0] A[3:0];
2  input reg [1:0] addr1, addr2;
3  input reg [15:0] datai;
4
5  always @ (posedge clock or posedge reset)
6  begin
7    if (addr1 >= addr2) begin
8      A[addr1 - addr2] <= datai;
9    end
10 end
```

The generated SMT expression from our tool, for transition from cycle 1 to cycle 2 would

be:

$$
\begin{aligned}
(A_0^2 = ite(addr1^0 \geq addr2^0, ite( & \\
& addr1^0 - addr2^0 = 0, datai^0, A_0^1), A_0^1 \\
& ) \wedge (A_1^2 = ite(addr1^0 \geq addr2^0, ite( \\
& addr1^0 - addr2^0 = 1, datai^0, A_1^1), A_1^1 \\
& ) \wedge (A_2^2 = ite(addr1^0 \geq addr2^0, ite( \\
& addr1^0 - addr2^0 = 2, datai^0, A_2^1), A_2^1 \\
& ) \wedge (A_3^2 = ite(addr1^0 \geq addr2^0, ite( \\
& \qquad addr1^0 - addr2^0 = 3, datai^0, A_3^1), A_3^1)
\end{aligned}
$$

Here $A_i^t$ represents $A[i]$ at cycle t, and is encoded in the SMT formula as 'A<i>___t'.

To perform bounded model checking of the RTL code, our tool takes all the SMT formula generated, replicates them for $k$ number of cycles after converting them into the appropriate version for cycle $i$. The reachability formula is for the target branch is generated by taking a conjunction of the activating and all preceding conditions of the branch. A disjunction of these formulas for various clock cycle is taken to form the final reachability assertion formula. After that, all of the circuit behavior formulas and the reachability assertion formula are given to Z3 SMT solver [20] using the provided C++ API.

### 3.3.2 Signal Domain Constraints

To perform the bounded model checking by itself, an initial state formula is generated with the reset sequences asserted and all register and memory initialized to the appropriate values. After which the solver is called to check the satisfiability of the model. If the solver returns

```
1  integer state;
2
3  always@(posedge clock) begin
4  if (reset)
5    state = 0;
6  else begin
7    case(state)
8      0: // branch: 1
9        state = 1;
10     1: // branch: 2
11       state = 0;
12     default: // branch: 3
13   endcase
14 end
```

Figure 3.1: Necessity for signal domain constraint for BMC without initial state demonstrated using variable 'state'

satisfiable for the formulas given, it can be deemed that the branch under scrutiny is definitely reachable. However, this method by itself cannot guarantee the unreachability of the target branch if the solver returns unsatisfiable. This method would only guarantee that the target branch is unreachable in $k$ number of clock cycles. In essence the number of branches shown to be unreachable is an upper bound for the number of unreachable branches in the circuit.

To ensure the unreachability of the branches without bound, we utilize BMC with k-induction. With this technique, after the bounded model checking in the base step, the induction step requires model checking without any initial state as described in Section 2.12. Normally the initial state for the induction step is unconstrained. But we need to only consider all the valid states after reset sequence of the circuit. Hence, we constrain the initial state of the induction step to only valid states represented by the signal domain constraints of the variables. For example, for the circuit described in Figure 3.1, branch 3 is unreachable starting from the initial state. The BMC assertion property for this branch with $k = 2$ will be generated as: $(state^1 \neq 0 \wedge state^1 \neq 1) \vee (state^2 \neq 0 \wedge state^2 \neq 1)$

This property will be unreachable by the base case of BMC. However, if the induction step is unconstrained for the initial state, the solver may pick $state^0 = 7$ arbitrarily. This value of $state^0$ will propagate to the variable on cycle 1 which will cause the induction step to incorrectly infer that branch 3 is reachable.

To prevent this, we add the signal domain constraints for all the variables involved. This ensures that the allowed starting variable values for the induction step is limited to a given set of values. Because the signal-domain analysis ignores all cycle information, the set of values obtained is conservative and an over-approximation of the actual set of attainable values.

For example, The signal domain constraint for the example described above is given by $\forall i, (state^i = 0) \vee (state^i = 1)$. Thus, on the induction step, the solver will be forced to assert state to 0 or 1 and eventually be able to infer the unreachability of branch 3.

### 3.3.3 Iterative Induction Step Using Partitioned Property

The nature of the induction step for BMC makes it less restrictive than the real world system. This has the side effect on many cases. One of this case is when considering nested branches. The nested branch's activating condition and the preceding conditions conjectured together make up the assertion property for BMC. For example in Figure 3.2 the assertion property $\phi$ as described in section 2.11 for branch 0, is given by $\phi = p$ and for branch 1, it is given by $\phi = p \wedge q$.

Now if branch 0 with the activating condition $p$ is unreachable, then $p$ will not be satisfiable in the base case. And during induction step, constraint $\neg p^1 \wedge \neg p^2 \wedge ... \wedge \neg p^k \wedge p^{k+1}$ will be added to the model. If the solver is not able to assert p to true in the $k+1$-th step, the solver will not be able to satisfy $p^{k+1}$ and we can infer branch 1 to be unreachable.

```
1 if(p) // branch 0
2   if(q) // branch 1
3     if(r) // branch 2
4       ...
```

| Iteration | Assertion Formula for Branch 2 |
|---|---|
| 1 | $p$ |
| 2 | $p \wedge q$ |
| 3 | $p \wedge q \wedge r$ |

Figure 3.2: Sample Assertion Formula on Different Iteration

In case of branch 1, which is nested inside, the activating condition $q$ may be easily satisfiable or unsatisfiable (e.g. if it is an input). In this case, base case would be unable to satisfy $p \wedge q$ as well. However, in the induction step, the constraint to be testing will be $\neg(p^1 \wedge q^1) \wedge \neg(p^2 \wedge q^2) \wedge ... \wedge \neg(p^k \wedge q^k) \wedge (p^{k+1} \wedge q^{k+1})$.

Since it can be easy for the solver to assert $q$ to any value, it can assert $\forall i, q^i \leftrightarrow \bot$. This will make the newly added constraint irrelevant as it will be always $\top$. Now, even though we know the outer branch to be unreachable, and by extension the inner loop to be unreachable as well, the k-induction technique was unable to prove the unreachability of that branch.

To get around this issue, we propose partitioning of the entire chain of activating and preceding conditions of a branch. This partition can be done using shared variables. If a condition and the preceding condition have the exact same variables, then we partition both of them into a same group.

For branches that could not be definitively inferred as reachable or unreachable, we apply this partitioning mechanism. Then, we apply the induction step of the BMC with each set of groups starting from the top most preceding condition. So on the first step only the first group's condition is used as the assertion condition and on the second step both first and second group's conditions are conjectured and so on. For example, in case of the case in 3.2, for branch 2, the partitions will be $p$, $q$, and $r$ if none of them have the same variables. With this partition, on first iteration, the induction step will be done with the assertion formula $p$, on the second iteration it will be done with $p \wedge q$ and so on. The detailed algorithm for

this is given in

**Algorithm** Iterative Induction Step on Partitioned Property

```
1: Solver BMC base case
2: Add Property φ for activating and all preceding conditions
3: Solve for Induction case
4: if base case unsat ∧ induction case sat then
5:     Z ← partition(φ)
6:     for i ∈ {1..n(Z) − 1} do
7:         for j ∈ {1..i} do
8:             φ ← φ ∧ Z[j]
9:         end for
10:        Add(φ)
11:        if sat then
12:            return "Unreachable"
13:            break
14:        end if
15:    end for
16: end if
```

## 3.4 Experimental Results

A tool implementing the methods described in the previous section was developed in C++. The input Verilog design is given to Verilator to generate a cycle-accurate C++ code as well as the AST of the C++ code. This AST file is parsed to get an internal interpretation of the C++ code. Program slicing is performed on this internal representation and the C++ API of Z3 SMT solver is used to generate the appropriate SMT formulas. These formulas are then solved using an instance of the Z3 solver.

Table 3.1 shows the number of unreachable branches found using different modes of the tool. For each circuit, the total number of branches is first reported. Next, the number of branches not reached by the vectors generated by BEACON [5] is shown. We also report the number of unreachable branches using the signal-domain analysis [35] alone in the next

**Algorithm** Bounded Model Checking

```
1: PGM SLICE(cycles = {1..k})
2: SSA INIT(cycles = {1..k})
3: P(cycles = {1..k})
4: PUSH(solver)
5: ADD(solver, reset constraints)
6: if satisfiable(solver) then
7:     return "Reachable"
8: else
9:     POP(solver)
10:    PGM SLICE(cycles = k + 1)
11:    SSA INIT(cycles = k + 1)
12:    P(cycles = k + 1)
13:    ADD SIGNAL DOMAIN CONSTRAINTS(solver)
14:    if satisfiable(solver) then
15:        return "Unknown"
16:    else
17:        return "Unreachable"
18:    end if
19: end if
```

column. The remaining columns report our method, with a given bound $k$. For example, for circuit b15, it contains 149 branches. BEACON vectors reached all but 16 branches. The signal domain analysis was able to prove 4 out of these 16 branches as unreachable. With $k = 2$, our BMC with induction was able to prove 9 branches to be unreachable. When iterative partitioning is added, we proved 10 branches to be unreachable. The induction with or without signal domain constraints and with or without property partitioning, the number of unreachable branches are a lower bound. A branch identified as unreachable is guaranteed to be unreachable. It can be seen that for all benchmark circuits apart from b15, all the branches not covered by the ATPG has been identified as unreachable branches. In other words, 100% coverage is obtained!

To show the scalability of our method, we first ran BEACON [5] on all of the benchmark circuits to rule out the branches that are reachable. We configured our tool to run after BEA-

| Bench | No. of Branches | Not reached by BEACON | Unreach Proved by [35] | k | Number of Unreachable Branches Found | | |
|---|---|---|---|---|---|---|---|
| | | | | | k-induction BMC | k-induction BMC with signal domain constraints | k-induction BMC with signal domain constraints and iterative partitioning |
| b06 | 24 | 1 | 1 | 2 | 1 | 1 | 1 |
| | | | | 5 | 1 | 1 | 1 |
| | | | | 10 | 1 | 1 | 1 |
| b07 | 20 | 2 | 1 | 2 | 1 | 2 | 2 |
| | | | | 5 | 1 | 2 | 2 |
| | | | | 10 | 1 | 2 | 2 |
| b10 | 32 | 1 | 1 | 2 | 1 | 1 | 1 |
| | | | | 5 | 1 | 1 | 1 |
| | | | | 10 | 1 | 1 | 1 |
| b11 | 33 | 1 | 1 | 2 | 1 | 1 | 1 |
| | | | | 5 | 1 | 1 | 1 |
| | | | | 10 | 1 | 1 | 1 |
| b13 | 64 | 4 | 4 | 2 | 2 | 4 | 4 |
| | | | | 5 | 2 | 4 | 4 |
| | | | | 10 | 2 | 4 | 4 |
| b14 | 211 | 14 | 12 | 2 | 2 | 14 | 14 |
| | | | | 5 | 2 | 14 | 14 |
| b15 | 149 | 16 | 4 | 2 | 9 | 9 | 10 |
| | | | | 5 | 9 | 9 | 10 |
| | | | | 10 | 9 | 9 | 10 |
| simple_spi | 65 | 1 | - | 2 | 1 | 1 | 1 |
| | | | | 5 | 1 | 1 | 1 |
| | | | | 10 | 1 | 1 | 1 |
| sasc | 72 | 6 | - | 2 | 6 | 6 | 6 |
| | | | | 5 | 6 | 6 | 6 |
| | | | | 10 | 6 | 6 | 6 |

Table 3.1: Number of Unreachable branches for BMC with or without initial state and using k-induction

CON on the branches that were unaccounted for. We show the run times for the branches in consideration in Table 3.2. We compare our run times against [35] to show that the overhead is minimal. We have split our run time for just the branches under consideration, because we intend our tool to be used as a way of explaining the unreachable branches after running test generation tools like BEACON, which is different from the intentions of [35].

| Bench | run time (s) [35] | branch ID | result of [35] | our result (k=5) | our run time (s) |
|---|---|---|---|---|---|
| b06 | 4.2 | 22 | Unreachable | Unreachable | 0.071 |
| b07 | 4.7 | 13 | Maybe Reachable | Unreachable | 0.169 |
| | | 18 | Unreachable | Unreachable | 0.186 |
| b10 | 6.5 | 30 | Unreachable | Unreachable | 0.238 |
| b11 | 12.9 | 31 | Unreachable | Unreachable | 0.132 |
| b13 | - | 14 | Unreachable | Unreachable | 0.037 |
| | | 27 | Unreachable | Unreachable | 0.185 |
| | | 37 | Unreachable | Unreachable | 0.169 |
| | | 59 | Unreachable | Unreachable | 0.295 |
| b14 | 63.8 | 105 | Unreachable | Unreachable | 24.09 |
| | | 185 | Unreachable | Unreachable | 20.79 |
| b15 | - | 14 | Unreachable | Unreachable | 3.35 |
| | | 114 | Maybe Reachable | Unreachable | 5.412 |
| | | 130 | Maybe Reachable | Maybe Reachable | 5.604 |
| | | 133 | Maybe Reachable | Unreachable | 3.39 |
| sasc | - | 37 | Unreachable | Unreachable | 0.060 |
| simple_spi | - | 38 | - | Unreachable | 0.533 |

Table 3.2: Run time comparison for different branches

## 3.5 Conclusion

In this work, we have presented a method to formally analyze a circuit and prove the unreachability of branches in the RTL code description of the circuit. This method utilizes program slicing to generate SMT formula of the circuit and Bounded Model Checking to prove the unreachability safety property of a given branch in certain number of cycles starting from the reset state. Furthermore, by utilizing signal domain constraints and k-induction with BMC, this method is capable of proving unreachability of branches irrespective of the number of cycles by losing accuracy in case of truly reachable branches.

# Chapter 4

# Sequential Test Vector Compaction using RTL Branch Coverage Metric

## 4.1 Introduction

For digital circuit testing on Automatic Test Equipment (ATE), it is important for the test size to be as small as possible. Large test sizes increase test time for each circuit which can be very expensive. However, with modern trends of large and complex circuits it is becoming increasingly difficult to come up with good or small tests. Every for moderately small circuits, the current approaches use heuristics to generate tests that are small. In other words, these approaches do not generate the optimum result.

The difficulty for generating small test sets rise from the enormous search space for the specific problem. Even for only combinational circuits to get the minimum possible test set that can reach all faults or all branches in RTL, the search space includes $2^n$ input vectors. For sequential circuits on the other hand, this space is much much larger. This makes the

problem of small sized test generation intractable. Hence, the most common approach is to generate vectors so that as many faults or RTL branches can be covered with as few vectors as possible [39].

There have been numerous approaches proposed for combinational test compaction. These approaches include fast gate level static compaction [40], dynamic compaction [41], independent and compatible fault fault sets based test generation [42, 43, 44], maximal compaction [43], rotating backtrace [43], double detection [44], forced pair merging [45], essential fault pruning [45], SAT-based solving algorithms [46], and partitioning of the scan flipflops to increase don't care bits [47].

The approaches of test compaction can be separated into two major types. These are static compaction and dynamic compaction. Static compaction [41] is the process where all independent tests are allowed to be generated first and then compaction operation is executed on these tests.

Another approach is to have a single small test cover as many faults as possible during the generation of tests. This approach is dynamic test compaction [41]. The generated tests themselves then can not be compacted any further after generation in this approach.

RTL Simulation based ATPG [5], [19] operates on RTL code under the assumption that RTL branch coverage is equivalent to fault coverage in the synthesized circuits. However often times the generated test vectors are not optimized for size. Hence, the application of these vectors on circuit in ATE may not be the most cost effective approach.

In this work, we propose a method for sequential test compaction using static compaction approach. We utilize the vector set output and internal information of simulation based ATPG BEACON to generate the minimum test with the same branch coverage as the original test. We use heuristic based approach to get a good enough non optimum result as the

optimum result is computationally intractable.

The main contributions of our work presented in this paper is as follows:

- We present an algorithm for compacting set of vectors generated by ant colony optimization base ATPG tool BEACON
- We present a way of storing metadata for set of vectors during test pattern generation, that can later be used to reconstruct a smaller vector set with same coverage.
- We describe an implementation of the presented approach and present the experiments of running BEACON on these benchmark and then utilizing the compaction technique to generate a significantly smaller test.

The rest of this paper is organized as follows. Section 4.2 introduces motivations behind the presented work. Section 4.3 presents the methodology in detail. The experimental results are discussed in Section 4.4. Finally, Section 4.5 concludes the chapter.

## 4.2 Motivation

Deterministic test generation techniques [38] can generate minimal possible tests to get maximum branch coverage. However, deterministic approaches are computationally expensive and sometimes it becomes infeasible to apply these methods on even small circuits. When applicable, these methods can restrict search space to low number of cycles to achieve this. However, restricting to a low number of cycles can often lead to increased generation time.

Simulation based methods [11, 19] can generate test vectors very fast. But often these tools do no infer anything about these generated tests. These tests are generated with a maximum number of cycles in mind. So often these tests can be very large in size. This approach may

be acceptable for design validation. However, post silicon verification would be extremely costly for this approach to be useful.

To tackle this issue of compact test generation, an interesting approach may be to use tests generated from simulation based ATPG techniques and use test compaction techniques to generate a small sized tests with the same RTL coverage as the original tests. The simulation based techniques usually generate a set of sequential test vectors each aimed a specific paths of the design. All of these test sets can be merged to get a small non optimum set of tests. This approach allows the generation of small tests quickly that is suited for ATPG.

## 4.3 Methodology

The approach we use to do test compaction starts with a run using an ant colony optimization based ATPG, namely BEACON. This approach instantiates multiple ants each with an initial set of random sequential test vectors of fixed length $N_r$. Then these vectors are used to do simulation on the circuit itself. This virtually represents the ants traversing the CFG. During this traversal, each ant deposits a virtual pheromone on the paths which is evaporated by a fixed coefficient on each tick.

After each round, the best ants are picked based on a score, mutated and the process is continued until no new path is found. Finally each remaining ant vectors are pruned to the last useful point and stitched together to generate the final test. However, during the test generation, no preference is given to an ant that can cover a branch early in the simulation to the ant that takes a longer time to reach that branch.

If each test set is saved, along with the point where each covers a new branch for the first time, then a set of vectors can be chosen that can cover all branches in the least number of

cycles. However, this approach will become intractable computationally. In fact if there are $n$ test sets for all ant throughout all iterations, then there can be ${}^nC_1 + {}^nC_2 + {}^nC_3 + ...{}^nC_n$ possible combinations of taking the test sets. As it can be seen, this naive solution can become extremely complex very soon.

In this section, first the storage of minimum cycle required for covering a branch using a particular test in a two dimensional matrix is described. Furthermore, a way of getting the optimum solution using an optimization solver is presented here, followed by a sub-optimum heuristics based approach.

### 4.3.1 Minimum Cycle Matrix

During ATPG using BEACON the minimum cycle matrix is constructed. The number of rows in this matrix is equal to the number of test sets generated and the number of columns is equal to the number of covered branches in the RTL code. As each new test is generated using ant evolution and mutation, the number of rows in this matrix grows. The value of the matrix is the first cycle at which the corresponding test set first reaches the corresponding branch. For example, in figure 4.1a consider the 4 test vectors generated. These vectors all have cycle length of 400. The numbers inside the test represent covering of a branch for the first time by that test. The first test covers branches 0, 1, 2 and 5 at cycles respectively and so on. In the minimum cycle matrix representation the generated matrix will be the matrix shown in 4.1b. Here the values $X$ represent that the particular branch is not covered by that test at all. The value 310 on second row, fourth column represents that the second test covers branch 3 first at cycle 310.

### 4.3.2 Optimum Solution using an Optimization Solver

To get the optimum solution, we propose using an optimization solver. Using the optimization solver, we describe a method for encoding the problem conditions. It is possible to get the optimum solution just using the minimum cycle matrix.

Each test set that has to be picked, has to be cut at a particular point. All of these sub tests extracted from the start of each test to the stated point will be joined to generate the final compacted test. If we assume that this cut point is denoted with $t_i$ for test number $i$, and that $t_i = 0$ means that test is not picked at all, then it can be stated that,

$$\forall i, t_i \geq 0$$

And if the maximum length of a test is $N_r$, then

$$\forall i, t_i \leq N_r$$

We can declare a new variable $p_{ij}$ where $p_{ij} = 1$ iff test number $i$ is used to cover branch number $j$ and $p_{ij} = 0$ if it is not. Then it can be written,

$$\forall i, \forall j, 0 \leq p_{ij} \leq 1$$

And since all Covered Branches must be picked,

$$\forall j, \sum_{i=0}^{N} p_{ij} \geq 1$$

where $N_b$ is the total number of covered branches. This works even in the case where single

test can cover multiple branches. As long as each branch is covered at least once by any test, this inequality will hold.

Now, picking branch $j$ using test set $i$ implies that the cut point for test $i$ has to be greater than or equal to the first cycle at which the test covered the branch. This information is available to use in the form of the $Q$ matrix. This logic can be encoded as,

$$(p_{ij} = 1) \rightarrow (t_i \geq Q_{ij})$$

Finally, we wish to make the final test as small as possible. Since the final test is just a concatenation of sub tests, each of which is of length $t_i$, then for the optimizer, the objective function will be

$$minimize : \sum_{i=0}^{Nt} t_i$$

where $Nt$ is the total number of tests under consideration.

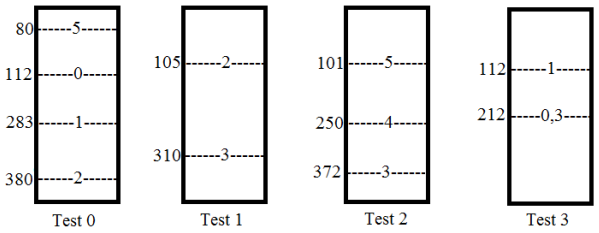
In our approach to get the optimum result, we feed all of these constraints to Z3 SMT solver configured as an optimization solver configured with pareto priority. The values for different $t_i$ is extracted from the model of the optimization solver and used to reconstruct the final test.

For example, for the scenario given in 4.1a and 4.1b the generated constraints will be:

$$\forall i, t_i \geq 0; i \in \{0, 1, 2, 3\}$$

$$\forall i, t_i \leq 400; i \in \{0, 1, 2, 3\}$$

$$\forall i, \forall j, 0 \leq p_{ij} \leq 1; i \in \{0, 1, 2, 3\}; j \in \{0, 1, 2, 3, 4, 5\}$$




(a) Four Tests Each with 400 Vectors Annotated with the first Cycle a Branch is Covered by it

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 112 & 283 & 380 & X & X & 80 \\ X & X & 105 & 310 & X & X \\ X & X & X & 372 & 250 & 101 \\ 212 & 112 & X & 212 & X & X \end{pmatrix} \end{array}$$

(b) The Minimum Cycle Matrix for the Given Four Tests and 6 Branches

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 112 & 283 & 380 & X \\ X & X & 105 & 310 \\ X & X & X & 122 \\ 212 & 112 & X & 212 \end{pmatrix} \end{array}$$

(c) The Minimum Cycle Matrix After Selection of First Test

$$\forall i, \sum_{j=0}^{N_b} p_{ij} \geq 1; i \in \{0,1,2,3\}; j \in \{0,1,2,3,4,5\}\ Q_{ij} \neq X$$

$$(p_{ij} = 1) \rightarrow (t_i \geq Q_{ij}); i \in \{0,1,2,3\}; j \in \{0,1,2,3,4,5\}\ Q_{ij} \neq X$$

And the minimization objective function will be:

$$minimize : \sum_{i=0}^{3} t_i$$

### 4.3.3 Heuristics Based Test Vector Compaction

In this subsection, we present a sub-optimal heuristics based approach for getting a small sized test by using all of the available tests. This approach first utilizes the concept of essential branches. These are the branches that are covered by one and only one test set. Hench to cover these branches, the corresponding test set have to be selected. With the selection of essential tests, the branches are covered by these tests within the number of cycles given by the $Q$ matrix are marked as covered. For the rest of the branches, this cycle number is subtracted from the $Q$ matrix.

It may be tempting to eliminate the test sets that are redundant in the next step. However, it would not help with the final goal of minimum test size. In other words, this would give the lowest number of test sets. However, each of these test set can be quite large. Hence, we would need to take into account the number of cycles these test set take to reach the respective branches. For example, picking two test set which cover a set of branches where each are 3 cycles long is better than picking one set which is 30 cycles long.

Instead of eliminating the redundant test set we apply heuristics based on the the number of cycles required by each test set to be able to cover a branch. It can be assumed that a branch that usually takes a large number of cycles to cover by most of the test set, but is covered in relatively low number of cycles by one test, then it is desirable to pick that branch, test set pair. We attribute a score to select such a branch given by,

$$score_j = \frac{\sum_i Q_{ij}/c_j - min_i(Q_{ij})}{\sum_{i,j} Q_{ij}}$$

where $c_i$ is the number tests that cover branch $j$.

With this score, the branch with the highest score is selected and the test set that covers

this branch in lowest cycle is picked. Here cut point is selected with the same low cycle as well. After that other branches that are covered using this test set within this clock cycle, are eliminated from further analysis as well. This is followed by subtracting the cycle from the $Q$ matrix row as described in the first step. After this, the essential branches, and the corresponding test set are picked again. This process is continued until all branches are picked.

For example, consider the case in 4.1a and the generated minimum cycle matrix in 4.1b. In the first step, it can be seen that to cover branch 4, test 2 has to be picked. So this test is picked. This test also covers branches 3 and 5. However, since we have to select upto cycle 250 to cover branch 4, branch 3 will not be automatically covered. So we mark branches 4 and 5 as done only and set $t_2 = 250$. We keep test 2 in place but subtract 250 from all of the elements in the entire row. If this test is later picked to cover any other branch, then it would be enough to just change the cut point of test 2, $t_2$ to whatever required at that time. The resulting minimum cycle matrix is shown in figure 4.1c

On the next step, since no more essential tests can be identified, the next test branch pair is selected using the score metric given previously. Using this score we get score for the remaining branches 0 to 3 to be 0.027, 0.046, 0.074, and 0.048 respectively. With these scores, branch 2 is chosen. Since is covered by test 1 in the lowest number of cycles, it is selected. This whole process continues until all branches are covered.

At the end of the selection process, we are left with $t_0 = 250$, $t_1 = 105$, $t_2 = 0$, $t_3 = 112 + 100 = 212$. So the final compacted test will be given by taking tests 0, 1 and 3 and the length of this compacted test will be 567. It has to be mentioned that reset sequences will need to be added between the sub tests before concatenation. The detailed algorithm is given in algorithm .

**Algorithm** Heuristic Based Test Vector Compaction

```
1: for all i ∈ All Tests do
2:     t_i ← len(reset seq)
3: end for
4: for all j ∈ covered not done branches do
5:     for all i ∈ All Tests do
6:         count ← 0
7:         if Q_ij ≠ NULL then
8:             k ← i;
9:             c ← Q_ij;
10:            count ← count + 1
11:        end if
12:        if count > 1 then
13:            break
14:        end if
15:        if count = 1 then
16:            t_k ← t_k + c
17:        end if
18:        Subtract c from all element in row k of Q
19:        for b ∈ covered not done branches do
20:            if Q_kb ≠ NULL ∧ Q_kb < c then
21:                Mark b as done
22:            end if
23:        end for
24:    end for
25: end for
26: for all j ∈ covered not done branches do
27:    s ← 0
28:    count ← 0
29:    for all i ∈ All Tests do
30:        s ← s + Q_ij
31:        count ← count + 1
32:        if Q_ij < l then
33:            l ← Q_ij
34:        end if
35:    end for
36:    D_i ← s/count − l
37: end for
```

38: **for all** $(j, d) \in D$ **do**
39: **if** $d > h$ **then**
40: $p \leftarrow j$
41: $h \leftarrow d$
42: **end if**
43: **end for**
44: **for all** $i \in$ All Tests **do**
45: **if** $Q_{ip} < l$ **then**
46: $l \leftarrow Q_{ip}$
47: $q \leftarrow i$
48: $c \leftarrow Q_{ip}$
49: **end if**
50: **end for**
51: $t_q \leftarrow t_q + c$
52: **for all** $b \in$ covered not done branches **do**
53: **if** $Q_{qb} \neq NULL \wedge Q_{qb} < c$ **then**
54: Mark b as done
55: **end if**
56: **end for**

## 4.4 Experimental Results

To test the effectiveness of our method for compaction of sequential test vectors generated by BEACON, we have implemented some add-on features to BEACON using C++. First of all, we generate a minimum cycle matrix as described in section 4.3.1. Along with growing this matrix as new test vectors are generated, we also keep a history of all the random functions in BEACON to be able to reconstruct these tests later. We implement two more add-on features which use this matrix to demonstrate approaches described in sections 4.3.2 and 4.3.3. The optimization tool uses Z3 SMT solver in linear integer arithmetic optimization mode using the Z3 C++ API.

The effectiveness of both the approaches is shown in table 4.1. The results shown in this table was done by fixing the initial seed of BEACON to a fixed value to ensure reproducability. The tests were done with a timeout of 15mins.

| Bench | Cycles | Ants | Reachable Coverage (%) | Number of Sequential Test Vecotrs | | | Run Time (ms) | |
|---|---|---|---|---|---|---|---|---|
| | | | | BEACON | Optimum | Heuristics Based | Optimum | Heuristics Based |
| b06 | 10 | 5 | 100 | 35 | 19 | 23 | 30 | $< 1$ |
| | 10 | 10 | 100 | 33 | 13 | 25 | 343 | $< 1$ |
| | 20 | 10 | 100 | 23 | 16 | 27 | 187 | $< 1$ |
| b07 | 30 | 5 | 100 | 95 | 43 | 87 | 16 | $< 1$ |
| | 100 | 5 | 100 | 103 | 43 | 46 | 18 | $< 1$ |
| | 100 | 10 | 100 | 103 | 43 | 47 | 146 | $< 1$ |
| b10 | 30 | 5 | 100 | 157 | 29 | 55 | 30 | $< 1$ |
| | 30 | 10 | 100 | 219 | 23 | 36 | 125 | $< 1$ |
| | 50 | 10 | 100 | 303 | 22 | 32 | 296 | $< 1$ |
| b11 | 5000 | 20 | 100 | 5430 | 3032 | 3035 | 671 | $< 1$ |
| | 5000 | 40 | 100 | 5559 | 2865 | 2868 | 1734 | $< 1$ |
| | 10000 | 20 | 100 | 5919 | 3032 | 3035 | 468 | $< 1$ |
| b13 | 10000 | 20 | 100 | 5092 | 2143 | 4291 | 797 | $< 1$ |
| | 10000 | 40 | 100 | 5411 | 2143 | 2221 | 2700 | $< 1$ |
| | 20000 | 40 | 100 | 4977 | 2143 | 2152 | 2260 | $< 1$ |
| b14 | 2500 | 10 | 100 | 7137 | 1314 | 1740 | 1236 | $< 1$ |
| | 2500 | 20 | 100 | 7419 | TO | 660 | TO | $< 1$ |
| | 5000 | 20 | 100 | 10220 | TO | 540 | TO | $< 1$ |
| b15 | 50000 | 10 | 88.59 | 13433 | TO | 4061 | TO | $< 1$ |
| | 10 | 10 | 88.59 | 11623 | TO | 3590 | TO | $< 1$ |
| | 20 | 10 | 88.59 | 8841 | TO | 6148 | TO | $< 1$ |

Table 4.1: The Test Vector Sizes for BEACON, the Optimal Compaction and Using the Heuristics Based Method

It can be seen that in all of the cases the heuristics based approach performs almost as good as the optimum solution possible using the available tests. However, in cases of larger circuits with hundreds of branches like b14 and b15, it can be seen that the optimum method timed out. Another interesting fact is that, when the coverage is low for these circuits, the heuristics based approach is no able to compress the test very much compared to when the coverage is very high. This is due to the fact that the tests to reach the hard to reach branches often cover a lot of other branches as well. Since these tests are favored in the heuristics based selection, the presence of these tests during high coverage helps the compression.

To show the scalability of the heuristics based method, it is run on similar set of tests beside the optimizer based approach. It can be seen that, in all of the cases the optimizer based approach is unable to scale well. Specifically when there are large number of branches, it fails to return a result in a reasonable time. The run time analysis of the optimization based system can be better seen from the plots show in figure 4.2 and 4.3. Here it can be seen that even though the run times are reasonable for low number of branches, less complex circuits and low number of tests, with increasing number of branches, complexity and number of tests, the run time for the optimizer based approach increases exponentially. Whereas, the heuristics based approach take insignificant time for completion.

## 4.5 Conclusion

In this chapter we have presented a technique for test vector compaction for sequential circuits. These test vectors are assumed to be generated during any kind of simulation based ATPG where compaction is not handled during pattern generation. By providing a mechanism for stating the problem as a optimization problem we showed one possible way for using an optimizer to get the optimum compaction from these vector set. Furthermore, we

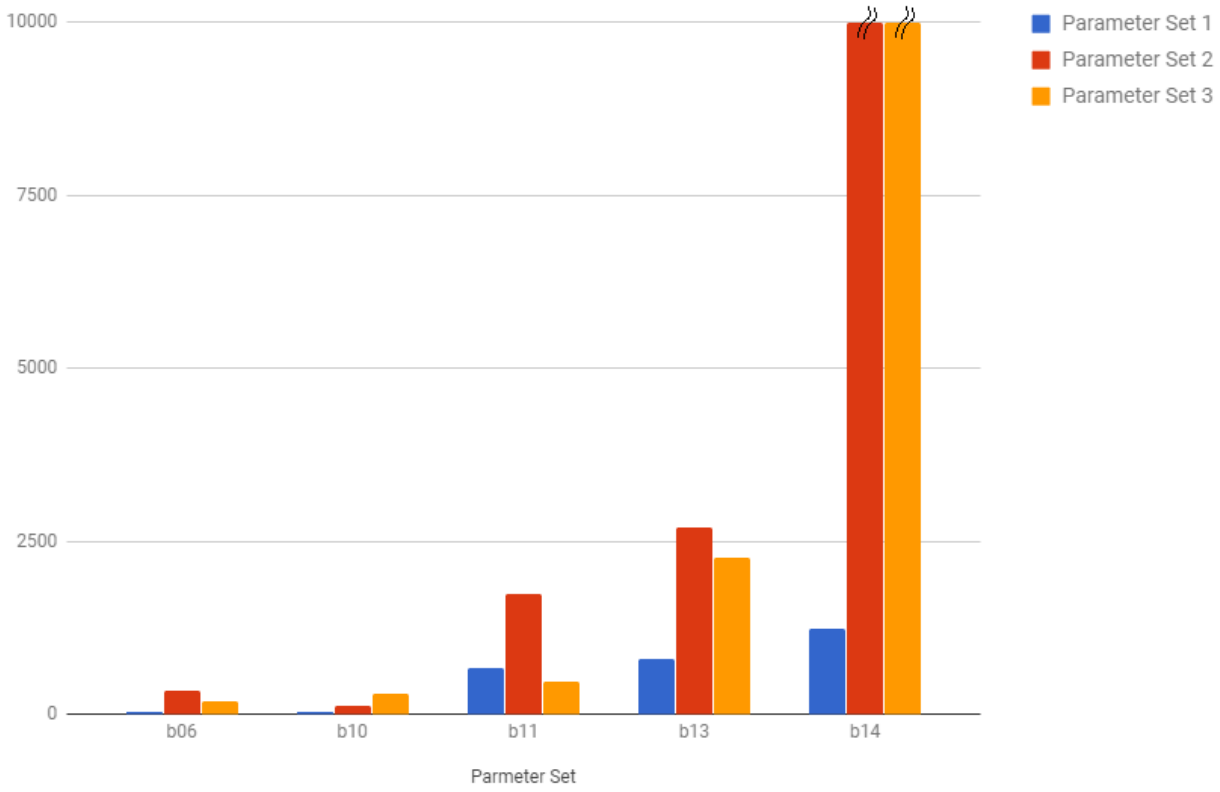



Figure 4.2: Run time for Optimizer Based Method for Different Benchmarks

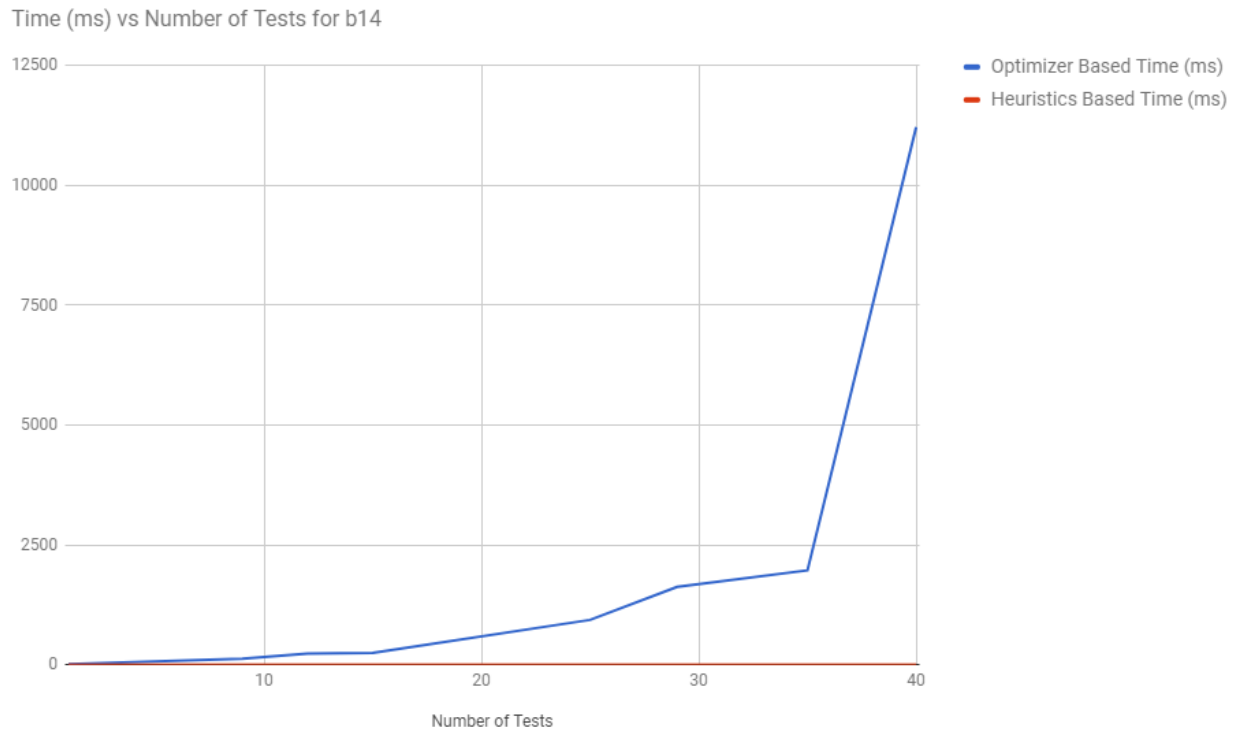



Figure 4.3: Run time vs Number of Tests for the two Approached for Benchmark b14

have presented a semi-optimal heuristic based approach. By comparing these two methods we showed with experimentation that the heuristics based approach is close to the optimal result, especially for large circuits requiring large number of cycles, while being magnitudes better in terms of time efficiency.

# Chapter 5

# Conclusions

We conclude the thesis by presenting concluding summary of the work presented. It is further appended with the limitations of the presented work and possible ideas for improving this work further in the future.

## 5.1 Concluding Summary

We have presented a method in Chapter 3 to prove the unreachability in RTL circuits using k-induction bounded model checking with signal domain constraints and property partitioning. This method was able to effectively prove the unreachability of previously unresolved branches.

Furthermore in Chapter 4 we have presented a method for generation of a compact sequential test set from the various tests generated in simulation based ATPG, BEACON without reducing coverage. This compaction mechanism is built using a heuristics that is also compared with an optimum solution using a linear integer optimizer. Our heuristics based approach

was proven to be comparably effective but magnitudes better in terms of time efficiency compared to the optimal solution.

## 5.2 Limitations and Future Work

While the unreachability method was effective at proving unreachability of all tested benchmark circuits, it was still unable to resolve some of the branches in the benchmark circuit b15. Moreover, the tests generated when a branch is proven to be reachable are underutilized in the present workflow. These tests can be used to guide the ATPG tools further to get better test patterns. Even when the base case of the k-induction is unsatisfiable, but the induction step is satisfiable, some insight can be obtained from the returned model that can be useful for test generation.

The heuristic based vector compaction mechanism was close to optimal for most of the cases. For some other case though, the performance could have been better. Better heuristics selection can be done to get rid of such cases. Moreover, the optimizer based compaction mechanism can be abstracted further. The optimizer based implementation can be tweaked to get a sub-optimal but more scalable system in future work

# Bibliography

[1] Gordon Moore. Progress in digital integrated electronics. *Electron Devices Meeting*, 21: 11–13, 1975.

[2] Harry Foster. The 2014 wilson research group functional verification study. 2014.

[3] Luciano Lavagno, Grant Martin, and Louis Scheffer. *Electronic Design Automation for Integrated Circuits Handbook - 2 Volume Set*. CRC Press, Inc., Boca Raton, FL, USA, 2006. ISBN 0849330963.

[4] B. Alizadeh and M. Fujita. Guided gate-level atpg for sequential circuits using a high-level test generation approach. In *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 425–430, Jan 2010. doi: 10.1109/ASPDAC.2010.5419843.

[5] M. Li, K. Gent, and M. S. Hsiao. Design validation of rtl circuits using evolutionary swarm intelligence. In *2012 IEEE International Test Conference*, pages 1–8, Nov 2012. doi: 10.1109/TEST.2012.6401556.

[6] A. Sen, J. Bhadra, V. K. Garg, and J. A. Abraham. Formal verification of a system-on-chip using computation slicing. In *2004 International Conference on Test*, pages 810–819, Oct 2004. doi: 10.1109/TEST.2004.1387344.

[7] Michael S. Hsiao, Elizabeth M. Rudnick, and Janak H. Patel. Sequential circuit test generation using dynamic state traversal. In *Proceedings of the 1997 European Conference on Design and Test*, EDTC '97, pages 22–, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7786-4. URL http://dl.acm.org/citation.cfm?id=787260.787647.

[8] M. S. Hsiao, E. M. Rudnick, and J. H. Patel. Application of genetically engineered finite-state-machine sequences to sequential circuit atpg. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(3):239–254, Mar 1998. ISSN 0278-0070. doi: 10.1109/43.700722.

[9] W. Wu and M. S. Hsiao. Efficient design validation based on cultural algorithms. In *2008 Design, Automation and Test in Europe*, pages 402–407, March 2008. doi: 10.1109/DATE.2008.4484714.

[10] A. Valmari. The state explosion problem. pages 429 – 528, Berlin, Germany, 1998//. state explosion problem;state space methods;computer aided analysis;concurrent systems verification;verification questions;advanced state space methods;computational complexity;.

[11] K. Gent and M. S. Hsiao. Functional test generation at the rtl using swarm intelligence and bounded model checking. In *2013 22nd Asian Test Symposium*, pages 233–238, Nov 2013. doi: 10.1109/ATS.2013.51.

[12] Y. Zhou, T. Wang, T. Lv, H. Li, and X. Li. Path constraint solving based test generation for hard-to-reach states. In *2013 22nd Asian Test Symposium*, pages 239–244, Nov 2013. doi: 10.1109/ATS.2013.52.

[13] Y. Zhou, T. Wang, H. Li, T. Lv, and X. Li. Functional test generation for hard-to-reach states using path constraint solving. *IEEE Transactions on Computer-Aided Design*

*of Integrated Circuits and Systems*, 35(6):999–1011, June 2016. ISSN 0278-0070. doi: 10.1109/TCAD.2015.2481863.

[14] J. Wang, H. Li, T. Lv, T. Wang, X. Li, and S. Kundu. Abstraction-guided simulation using markov analysis for functional verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(2):285–297, Feb 2016. ISSN 0278-0070. doi: 10.1109/TCAD.2015.2419622.

[15] Frank Vahid. *Digital Design with RTL Design, Verilog and VHDL*. Wiley Publishing, 2nd edition, 2010. ISBN 0470531088, 9780470531082.

[16] Ieee standard for verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 0_1–560, 2006. doi: 10.1109/IEEESTD.2006.99495.

[17] Ieee standard vhdl language reference manual. *IEEE Std 1076-2000*, pages i–290, 2000. doi: 10.1109/IEEESTD.2000.92297.

[18] Verilator. https://www.veripool.org/wiki/verilator. [Online; accessed 15-July-2017].

[19] L. Liu and S. Vasudevan. Efficient validation input generation in rtl by hybridized source code analysis. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011. doi: 10.1109/DATE.2011.5763253.

[20] Z3. https://github.com/Z3Prover/z3. [Online; accessed 15-July-2017].

[21] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970. ISSN 0362-1340. doi: 10.1145/390013.808479. URL http://doi.acm.org/10.1145/390013.808479.

[22] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003. ISBN 1558607242.

[23] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996. ISBN 0818673842.

[24] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 1-55860-286-0.

[25] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: 10.1145/73560.73561. URL http://doi.acm.org/10.1145/73560.73561.

[26] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991. ISSN 0164-0925. doi: 10.1145/115372.115320. URL http://doi.acm.org/10.1145/115372.115320.

[27] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4): 352–357, July 1984. ISSN 0098-5589. doi: 10.1109/TSE.1984.5010248.

[28] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19(1):7–34, July 2001. ISSN 0925-9856. doi: 10.1023/A:1011276507260. URL https://doi.org/10.1023/A:1011276507260.

[29] S. Deng, W. Wu, and J. Bian. Bounded model checking for rtl circuits based on algorithm abstraction refinement. In *2006 8th International Conference on Solid-State and Integrated Circuit Technology Proceedings*, pages 2082–2084, Oct 2006. doi: 10.1109/ICSICT.2006.306623.

[30] Liang Zhang, M. R. Prasad, and M. S. Hsiao. Incremental deductive inductive reasoning for sat-based bounded model checking. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pages 502–509, Nov 2004. doi: 10.1109/ICCAD.2004.1382630.

[31] Niklas En and Niklas Srensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543 – 560, 2003. ISSN 1571-0661. BMC'2003, First International Workshop on Bounded Model Checking.

[32] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. *Checking Safety Properties Using Induction and a SAT-Solver*, pages 127–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. ISBN 978-3-540-40922-9.

[33] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software verification using k-induction. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 351–368, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23701-0. URL http://dl.acm.org/citation.cfm?id=2041552.2041578.

[34] Mikhail Y. Gadelha, Hussama I. Ismail, and Lucas C. Cordeiro. Handling loops in bounded model checking of c programs via k-induction. *Int. J. Softw. Tools Technol. Transf.*, 19(1):97–114, February 2017. ISSN 1433-2779. doi: 10.1007/s10009-015-0407-9. URL https://doi.org/10.1007/s10009-015-0407-9.

[35] S. Bagri, K. Gent, and M. S. Hsiao. Signal domain based reachability analysis in rtl

circuits. In *Sixteenth International Symposium on Quality Electronic Design*, pages 250–256, March 2015. doi: 10.1109/ISQED.2015.7085434.

[36] Michael S. Hsiao, Elizabeth M. Rudnick, and Janak H. Patel. Dynamic state traversal for sequential circuit test generation. *ACM Trans. Des. Autom. Electron. Syst.*, 5(3): 548–565, July 2000. ISSN 1084-4309. doi: 10.1145/348019.348288. URL http://doi.acm.org/10.1145/348019.348288.

[37] D. Krishnaswamy, M. S. Hsiao, V. Saxena, E. M. Rudnick, J. H. Patel, and P. Banerjee. Parallel genetic algorithms for simulation-based sequential circuit test generation. In *Proceedings Tenth International Conference on VLSI Design*, pages 475–481, Jan 1997. doi: 10.1109/ICVD.1997.568180.

[38] Lingyi Lui and Shobha Vasudevan. *STAR: Generating input vectors for design validation by Static analysis of RTL*, pages 32–37. 11 2009. ISBN 9781424448234. doi: 10.1109/HLDVT.2009.5340179.

[39] D.S. Hochbaum. An optimal test compression procedure for combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15 (10):1294 – 9, 1996/10/. ISSN 0278-0070. URL http://dx.doi.org/10.1109/43.541449. test compression procedure;combinational circuits;test vectors;set cover problem;fault ordering;integer programming techniques;.

[40] M. S. Hsiao, E. M. Rudnick, and J. H. Patel. Fast static compaction algorithms for sequential circuit test vectors. *IEEE Transactions on Computers*, 48(3):311–322, Mar 1999. ISSN 0018-9340. doi: 10.1109/12.754997.

[41] Elizabeth M. Rudnick and Janak H. Patel. Efficient techniques for dynamic test sequence compaction. *IEEE Trans. Comput.*, 48(3):323–330, March 1999. ISSN 0018-9340. doi: 10.1109/12.754998. URL http://dx.doi.org/10.1109/12.754998.

[42] S.B. Akers, C. Joseph, and B. Krishnamurthy. On the role of independent fault sets in the generation of minimal test sets. pages 1100 – 7, Washington, DC, USA, 1987//. logic testing;independent fault sets;minimal test sets;automatic test generation;global characterization;testing requirements;benchmark circuits;.

[43] Irith Pomeranz, Lakshmi N. Reddy, and Sudhakar M. Reddy. Compactest: A method to generate compact test sets for combinational circuits. pages 194 – 203, Nashville, TN, USA, 1992. Benchmark circuits;COMPACTEST;.

[44] S. Kajihara, I. Pomeranz, K. Kinoshita, and S.M. Reddy. Cost-effective generation of minimal test sets for stuck-at faults in combinational logic circuits. pages 102 – 6, Baltimore, MD, USA, 1993//. cost effectiveness;minimal test sets;stuck-at faults;combinational logic circuits;heuristics;independent fault sets;target faults;.

[45] Jau-Shien Chang and Chen-Shang Lin. Test set compaction for combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(11):1370 – 8, 1995/11/. ISSN 0278-0070. URL http://dx.doi.org/10.1109/43.469663. test set compaction;ISCAS'85 benchmark circuits;essential fault pruning;incompatible specified bits;forced pair-merging;essential faults;active compaction methods;combinational circuits;.

[46] S. Eggersgluss, K. Schmitz, R. Krenz-Baath, and R. Drechsler. On optimization-based atpg and its application for highly compacted test sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(12):2104 – 17, 2016/12/. ISSN 0278-0070. URL http://dx.doi.org/10.1109/TCAD.2016.2552822. optimization-based ATPG;test compaction;post-production test;test data;test costs;automatic test pattern generation;optimization satisfiability-based

ATPG;multiple-target test generation;optimization techniques;stuck-at as;industrial circuit;.

[47] Boxue Yin, Dong Xiang, and Zhen Chen. New techniques for accelerating small delay atpg and generating compact test sets. pages 221 – 226, New Delhi, India, 2009. URL http://dx.doi.org/10.1109/VLSI.Design.2009.64. Broad-side scan testing;Cpu time;Enhanced scans;Scan chains;Scan flip-flops;Scan-based tests;Small delay defect;Target faults;Test compactions;Test patterns;Test sets;Test-data volumes;The longest testable path selection;.