Improving Branch Coverage in RTL Circuits with Signal Domain Analysis and Restrictive Symbolic Execution

Sharad Bagri

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Michael S. Hsiao, Chair
A. Lynn Abbott
Chao Wang

February 2, 2015
Blacksburg, Virginia

Keywords: RTL circuits, Validation, Verification, Reachability, Unreachable, Branch Coverage, Signal Domain, Test Generation, Symbolic Execution

Copyright 2015, Sharad Bagri

Improving Branch Coverage in RTL Circuits with Signal Domain Analysis and Restrictive Symbolic Execution

Sharad Bagri

ABSTRACT

Considerable research has been directed towards efficient test stimuli generation for Register Transfer Level (RTL) circuits. However, stimuli generation frameworks are still not capable of generating effective stimuli for all circuits. Some of the limiting factors are 1) It is hard to ascertain if a branch in the RTL code is reachable, and 2) Some hard-to-reach branches require intelligent algorithms to reach them.

Since unreachable branches cannot be reached by any test sequence, we propose a method to deduce unreachability of a branch by looking for the possible values which a signal can take in an RTL code without explicit unrolling of the design. To the best of our knowledge, this method has been able to identify more unreachable branches than any method published in this domain, while being computationally less expensive.

Moreover, some branches require very specific values on input signals in specific cycles to reach them. Conventional symbolic execution can generate those values but is computationally expensive. We propose a cycle-by-cycle restrictive symbolic execution that analyzes only a selected subset of program statements to reduce the computational cost. Our proposed method gathers information from an initial execution trace generated

by any technique, to intelligently decide specific cycles where the application of this method will be helpful. This method can hybrid with simulation-based test stimuli generation methods to reduce the cost of formal verification. With this method, we were able to reach some previously unreached branches in ITC'99 benchmark circuits.

Dedicated to my grandmother, Baiji,

for her love, support and teachings

Acknowledgements

God makes everyone talented in some fields, but for the talents to materialize in an appreciable form, it requires the support of other brilliant souls. My work is no different. This thesis and the related publications may bear only my name, but the truth is, it was tremendously improved because of the guidance, help, support and love of many wonderful people. They deserve a note of immense thanks in the beginning pages of this work.

Dr. Michael Hsiao, my advisor, has been a wonderful mentor. His patient guidance and prompt replies has significantly enhanced this research work. I greatly appreciate his mentorship skills such as always motivating to innovate, giving much freedom to experiment, quickly finding shortcomings in ideas, and giving expert guidance to improve it. With someone like him as an advisor, I feel confident to be able to do research in any field. Thank you Dr. Hsiao, you will always be an inspiration for me.

I would also like to thank Dr. Chao Wang and Dr. A. Lynn Abbott for being on my committee and for all the help.

I would also like to thank my wonderful lab members for making the lab a nice place to work. They include: Sarvesh, for helping me in so many things in the lab and in general; Kelson, Vineeth, and Michael for proofreading my writings and the excellent research discussions; Arijit, Hassan, Markus, Sarmad, Avinash, Indira, Dilip, and Shuchi for the numerous interactions and help in the lab.

Many people contributed in making my stay in Blacksburg enjoyable. Thanks a lot to Ashutosh, Mahesh, Nikhil, Robin, Vijeyendra, Santhip, Ashok, Himanshu, Deepak, Vireshwar, Dinesh, Pooja, Chosang and his family, and many more people. Thanks a lot to my friend, Ivy, for being an inspiration to me in so many ways.

Even before I started graduate studies, many people back home had put a lot of effort to make it happen. Immense thanks to Mukund Mamaji, Shakuntala Mamiji, Giriraj Kaku, Kakima, Hari Mamaji, Usha Mamiji, Abhishek Jijaji, Bahan, Shiv Dadaji, Kashi Dadiji and R. D. Sethna foundation for helping me to come to the USA and study at Virginia Tech.

I am highly indebted and grateful to my father, Mr. Vijay Bagri, my mother, Mrs. Asha Bagri, and my sisters Ritu and Varsha for always supporting and motivating me. Thanks to all my family members, especially Vallabh Bhaiya, Narmada Bhabhi, Keshavji and Shruti for all the help.

Most importantly, I am grateful to God for everything given to me in this blessed life.

Sharad Bagri

Blacksburg, Virginia

January 2015

Attribution

My colleague, Kelson Gent helped me with one of the chapters presented as part of this thesis. Kelson is doing his PhD in Computer Engineering at Virginia Tech, Blacksburg. He helped me with literature review about the previous research work and methodology presented in chapter 3 of this thesis.

My advisor, Dr. Michael Hsiao also helped me with the work presented in this thesis. Dr. Hsiao is a professor in The Bradley Department of Electrical and Computer Engineering at Virginia Tech, Blacksburg. He helped me with brainstorming and reviewing the ideas discussed presented in chapter 3 of this thesis.

Contents

1 Introduction 1
1.1 Problem Scope and Motivation 1
1.2 Contributions of the Thesis 3
1.3 Publications 5
1.4 Thesis Organization 6
2 Background 7
2.1 Testing of Digital Circuits 7
2.2 Register Transfer Level description 10
2.3 Verilator 12
2.4 Activating and preceding conditions 13
2.5 Satisfiability Modulo Theory 14
2.6 Source to Source transformation 16
2.7 Static Single Assignment 17
3 Signal Domain based Reachability Analysis in RTL circuits 19
3.1 Overview 19

3.2 Introduction 20

3.3 Motivation and Related Work 21

3.4 Methodology.. 23

3.5 Some Special Cases 34

3.6 Application of this work 39

3.7 Comparison with other methods.. 41

3.8 Results 43

3.9 Analysis of unreached coverages 46

3.10 Conclusions 47

4 Reaching Branches in RTL code with Cycle-By-Cycle Restrictive Symbolic Execution 48

4.1 Overview 48

4.2 Introduction 49

4.3 Motivation and Related Work 51

4.4 Preliminaries.. 52

4.5 Methodology... 56

4.6 Results 63

4.7 Application of our work 65

4.8 Conclusions 66

5 Conclusion and Future Research Ideas 67

6 Bibliography 70

List of Figures

Figure 2.1: Sequential circuit unrolling 8

Figure 2.2: Verilog Code and its equivalent code in C++ converted code by Verilator .. 12

Figure 2.3: Sample Verilated C++ code explaining branch numbers and different conditions 13

Figure 3.1: Example Verilated C++ code showing only important part of the program .. 24

Figure 3.2: Assignment graph for the example code above... 24

Figure 3.3: Detailed output when Signals in Conditions are input 30

Figure 3.4: Detailed output when Signals in Conditions are assigned constant values 31

Figure 3.5: Detailed Output when signals in conditions are assigned values which are signals 32

Figure 3.6: Output of short format 33

Figure 3.7: Non-blocking assignment code in verilated RTL function 35

Figure 3.8: A sample output of our method for an unreachable branch 40

Figure 3.9: An example RTL circuit representation for comparing with our method 42

Figure 4.1: Example Verilated C++ program 54

Figure 4.2: Code Hierarchy Tree for verilated code of Figure 4.1 54

List of Tables

Table 3.1: Details about resolution of unreachable branches by our method 43

Table 3.2: Run time for different levels of recursion for all branches for our method..... 45

Table 4.1: Result of application of our method on short and long random vector sequence 64

Table 4.2: Result of application of our method for branches unresolved by BEACON for b14... 65

1 Introduction

Electronic chips make up the foundation of modern day computing and are pervasive around us in the form of various gadgets, machines and appliances. They aid us in tasks, from the trivial, such as making a cup of coffee, to our audacious endeavors to push the limits of our knowledge, such as sending a mission to Mars. They play an increasingly larger role in adding to the intelligence, which even the tiniest of electronic devices now possess. Many of those chips run critical applications, like the automatic brake system in cars, or the pacemaker in a heart. Any malfunction of those chips can lead to serious and irrevocable losses, or even loss of a precious life. With pervasive use in critical applications, it is imperative to verify them before they are put to actual use.

1.1 Problem Scope and Motivation

Design verification ascertains that the design meets the specification. In the beginning, only few transistors and other components could be placed on the electronic chips. The chips had very simple functionalities [1] and could be verified by manual inspection. The advances over the years in Intergrated Circuit (IC) design have made it possible to put more components into

small area of the chip. This has enabled us to put more real world complex logics on those chips without reaching to the limits of area or other constraints. This has definitely made us much better in delegating our tasks to those little chips, but the chips themselves have become huge in terms of the logic and number of transitors inside them. Verifying such giant designs is not an easy task, definitely not as easy as in the past, because of their increasing complexity.

As more components are added, verifying the complete design has become increasingly difficult. Similarly, the complex logic of individual modules make the verification of individual design modules also more challenging. This has led to increased importance of verification in the chip development process [2]. Currently, at least 50% of chip design time is spent on verification efforts[3], making new breakthroughs in verification much more urgent.

Design verification needs to be done at each level of the design process. While formal verification can effectively check if the implementation conforms to the spec without simulating any single vector, its practicality to date is still quite limited. Thus, the practical approach to verify a design is through the application of sequences of inputs, also called vectors or test patterns, to the circuit, and the output of the design is compared with the expected values. The information at one level of the design can aid in test generation at other levels. Particularly, Register Transfer Level (RTL) of design description has information such as “Don’t cares” (DC), valid states of each signal etc., which is lost at the circuit and physical design levels. This high level information can aid in generation of efficient test vectors at lower levels [4].

Today’s complex circuits have large number of inputs. These circuits cannot be exhaustively tested because there are numerous possibile combinations of inputs. For example, a circuit with 50 primary inputs has 1 thousand-trillion possible combinations. Applying each of them would be prohibitively expensive. Furthermore, the sequencing of the vectors play critical

roles in sequential circuits. So the number of possible sequences is much larger than one thousand-trillion. For a higher numbers of inputs, applying so many input vectors one by one would be infeasible, as the time required to apply those vectors would be in years, even with the fastest testers.

Over the years, there has been much research in generating effective stimuli for functional verification. These methods fall in the category of either simulation or deterministic methods. All the published works related to functional verification have not been able to achieve 100% coverage [5].

This thesis surpasses the work done by the previous papers in this field. The first contribution of the thesis is a methodology that deteremines the reachability of each branch of the circuit's RTL code. The second contribution of the thesis is to generate input vector for few branches which have not been reached earlier. In the process, it proves few more branches as unreachable. This work suggests vectors and proves unreachability for branches which have not been reached by earlier papers on a few benchmark circuits. Thus, for some of the circuits it takes the knowledge of reachability of all branches of the RTL code to 100%.

1.2 Contributions of the Thesis

The two research contributions of this thesis are as follows

1.2.1 Development of Signal Domain based Reachability Analysis in RTL circuits

This work proposes a method to determine the reachability of each branch in an RTL code written in Verilog. The domain of a signal, s, refers to all the values which s can take in the code if all the statements inside the code were considered to be reachable. This method performs

an analyis based on the domain of every signal. Because it does the analysis without unrolling the circuit, it has been able to prove unreachability of branches which could not be proven by traditional methods like Bounded Model Checker (BMC), Cone-of-influence, Bounded cone-of-influence with fixed time-bounds. The proposed analysis is quick and easy as it does not require the circuit to be unwound for a large number of cycles. For the branches it recommends as unreachable, it is guaranteed to be unreachable. For the reachable branches, it recommends the statements inside the code whose execution would facilitate reaching that reachable branch. This method is recursive and can go deeper in the analysis to recommend paths to the target branch. This path can be thought of as a concatenation of statements that flow the information towards the target branch. However, all the paths which this method recommends are not guaranteed to be true paths.

Thus, this information can be used as heuristic for a functional vector generation framework. The functional vector generation framework can try executing each of the paths in order to possbily reach the target branch.

1.2.2 Reaching Branches in RTL code with Cycle-By-Cycle Restrictive Symbolic Execution

In the second contribution, a modified version of symbolic execution is applied on certain cycles of execution trace to analyze the unreached branches. These unreached branches include branches which have not been proved unreachable by the first contribution, as well as by any other published research. In this method, the execution trace of the RTL code is examined for the branches reached in each cycle. From the cycles where a set of identified branches is reached, a set of cycles is determined where the target branch has a high probability of being reached. For each of those cycles, the statements executed inside the code are extracted in the order in which

they were executed in the trace. A static analysis is performed on the code which in some cases proves the unreachability of certain branches of the code. If the static analysis returns a favorable chance of reaching the target branch, then further constraints are added to the analysis to get the exact values of the input which would facilitate the reaching of the target branch.

The value suggested by the restricted symbolic analysis is the value which should be applied to the input of the circuit for the suggested cycle in order to reach the target branch. We verified the results, by independently executing the suggested inputs and found that every time the target branch was reached. A few of those branches were never reached by any other paper.

1.3 Publications

The publications related to this thesis are listed below:

- [6] **Sharad Bagri**, Kelson Gent, Michael S. Hsiao "Signal Domain Based Reachability Analysis in RTL Circuits," Proceedings of International Symposium on Quality Electronic Design (ISQED), March 2015.
- **Sharad Bagri**, Vineeth V. Acharya, Michael S. Hsiao "Reaching Branches in RTL code with Cycle-By-Cycle Restrictive Symbolic Execution" under review at GLSVLSI' 15.

Other research papers that were published and submitted during this research are as follows:

- Vineeth V. Acharya, **Sharad Bagri**, Michael S. Hsiao "Branch Guided Functional Test Generation at the RTL" submitted to ETS'15, under review
- [7] Sarvesh Prabhu, Vineeth V. Acharya, **Sharad Bagri**, Michael S. Hsiao "Property-checking based LBIST for improved diagnosability", 19th IEEE European Test Symposium (ETS), pp: 1-2, Paderborn, Germany, May 2014

- Sarvesh Prabhu, Vineeth V. Acharya, **Sharad Bagri**, Michael S. Hsiao "A diagnosis-friendly LBIST architecture with property checking", in proceedings of the IEEE International Test Conference (ITC), October 2014

1.4 Thesis Organization

The rest of the thesis is organized as follows.

- Chapter 2 describes the background concepts used in explaining this research work. It involve topics like testing of digital systems, SMT solver, tools, and concepts used in this research.
- Chapter 3 describes the work of signal domain based reachability analysis.
- Chapter 4 describes the work of cycle-by-cycle restrictive symbolic execution on RTL circuits.
- Chapter 5 concludes the thesis and provides ideas for future work.

2 Background

In this chapter, we present the details of the testing of digital circuits, details of RTL representation of circuits, various concepts such as Satisfiability Modulo Theory (SMT), Static Single Assignment (SSA) and tools such as Verilator, z3, pycparser which have been used for this research.

2.1 Testing of Digital Circuits

Circuits can be divided in two categories: combinational and sequential. Sequential circuits are circuits with memory elements, such as flip-flops, that can save register values in between cycles. Combinational circuits can't save any value between cycles.

In order to facilitate understanding of sequential circuits, they are modelled as copies of the same circuit over many time frames or cycles. This is called sequential circuit unrolling and an example of it is presented in Figure 2.1.

In the figure time frame number is marked as "TF number". The big rectangular block in time frame 0 and the big dots in time frame -3, -2, 1 and 2 represent a copy of the same

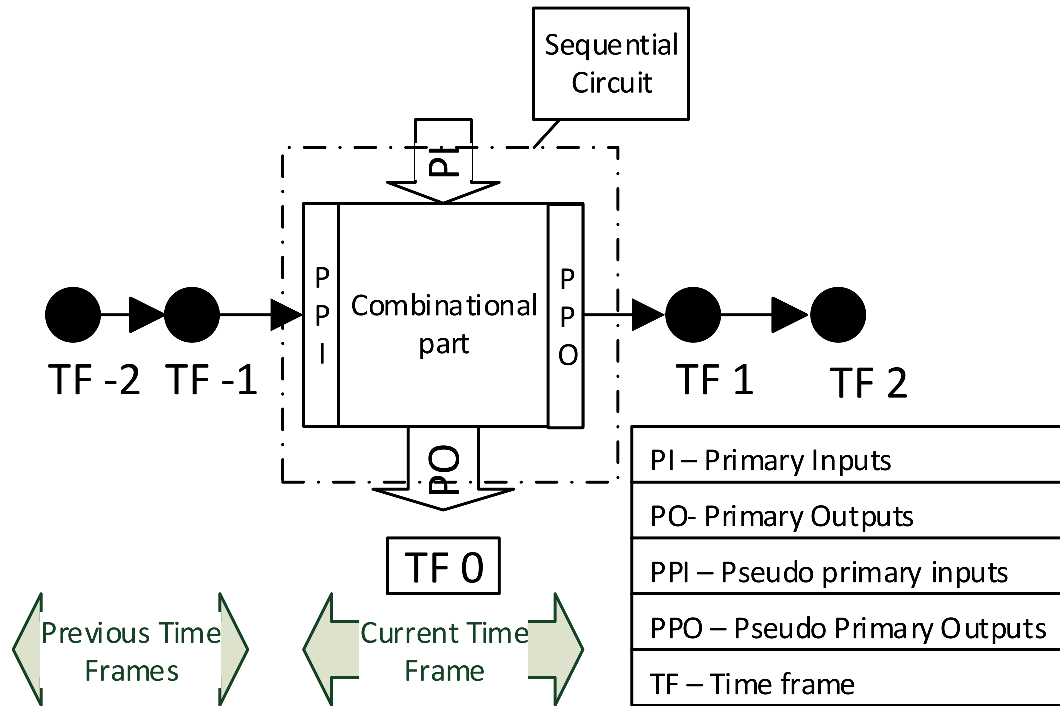

Figure 2.1: Sequential circuit unrolling

sequential circuit in respective cycles. This unrolling is not limited to the time frames presented in the figure. They can be extended to more cycles as needed. The horizontal arrows represent the flow of information from one cycle to next. The vertical arrow marked as PI represents primary inputs while the vertical arrow marked marked as PO represents the primary output of the circuit. The cycle under consideration is called current cycle or current time frame and is conventionally given a cycle number 0. while the cycles before current cycle are called previous cycles or previous time frames.

In unrolling, the memory elements can be modelled separately from the combinational part of the circuit. This has the advantage of clearly demarcating the operations between the cycles and during the cycle. Pseudo Primary Inputs (PPIs) and Pseudo Primary Outputs (PPOs) are the memory elements typically implemented using flip flops. The PPOs save value at the end of the cycle that are assigned to PPIs of the next cycle.

To test one fault or a property in a combinational circuit, just one input value needs to applied. However, in sequential circuits the value of flip flops or the state needed to test a fault needs to be justified from previous state or initial state which may require several cycles. This set of inputs to be applied sequentially over multiple cycles to test a property is referred as test vector or test stimulus.

Due to the ability of saving states, sequential circuits are more useful than combinational circuits as they can implement state machines. Hence they are found in most of the electronic designs of today. A state machine can have some states which can be reached only when a particular order of states is reached earlier. Such states are difficult to reach by random test vectors. Hence, these states are hard-to-reach. To test these hard-to-reach states may require a long sequence of specific test vector. To fully verify a chip these hard-to-reach states must also be verified. Sometimes one hard-to-reach state can unlock several other states which are easy to reach, starting from that particular hard-to-reach state. As test vectors to reach the hard-to-reach states may be very long, generating them by using deterministic methods such as Satisfiability (SAT) or Satisfiability Modulo Theories (SMT) solvers (described in section 2.5) are not feasible because the solution's state space can easily explode. Use of SMT solvers for few cycles has been reported in STAR [8] and state space explosion was a reason for its limitation.

A simulation based method, such as ant colony optimization (ACO) used in BEACON [9], has shown good results. Both of these, deterministic and simulation based methods, lie on opposite extremes of their approaches. One uses a deterministic solver which can't go back or unroll to a high number of cycles. Even solving for ten cycles has been reported to be infeasible for moderate-sized circuits[10]. The other category uses a simulation-based technique where guided random vectors are generated and they are checked for their fitness based on some metric.

One potential drawback with simulation based approach is that it does not learn any of the circuit features as it progresses in generating the vector. It has shown good coverage in some cases; however, some heuristics had to be hard coded for it.

Pure simulation based methods may not know which branches are unreachable and can spend considerable effort in order to reach them. They would ultimately fail, leading to wasted efforts. In this regard it would be helpful to know which brachnes are unreachable before the test vector generation program starts generating test vector, so that, such waste of effort can be avoided.

2.2 Register Transfer Level description

Electronic design proceeds in a layered fashion, proceeding from high level descriptions which are better understood by humans, to eventually the low level transistor layout details that are needed for fabrication. Register Transfer Level (RTL) description is a design abstraction which defines the behavior of the circuit without going into the implementation details of it. It describes the circuit in terms of signals, memory elements and logical operations performed on them. Hardware Description Languages (HDLs) like Verilog or VHDL are used to describe it.

Verilog or VHDL code is similar to code written in high level languages like C++, Java etc. Codes in C++ don’t delve into the hardware implementation details such as which registers to use, which buses to use etc. Similarly, HDLs don’t go in the details of which transistors should be used for which modules. Hence, programs written in HDL and C++ are conceptually similar. However, as HDLs implement hardware which in some matters is different from software, HDLs have additional constructs to handle such differences.

In hardware, all the modules run in parallel, so a serial program executing one line at a time is not appropriate to model it. For example, on a chip, if there are 10 input signals then all

10 inputs will receive values in parallel in a particular cycle. However, in software typically it would be sequential, where only one signal would be updated at a particular instance.

These differences sometimes create significant differences in signal values, if they are not modelled appropriately by the programming language used. To take care of this difference between hardware and software, Verilog has two kinds of assignments namely

1. Blocking assignment: evaluation and assignment are immediate in assignments of this type. Example: `a = b + c`. In this code, `a` will have updated value of `b + c` before moving to next line in the code. This is what traditionally happens in all high level languages like C++, Java etc.
2. Non-Blocking assignment: In this kind of assignment all right-hand side expressions are evaluated and then the assignments are done. Example: if there are lines as `a <= b + c; b <= c | a; c <= a ^ b;` then the operations on the right hand side of all three lines will be done first before assigning values to the signals on left hand side. It has the effect of updating the values of the signals only at the end of the clock cycle.

Though it may look that both kind of assignments should produce same result but sometimes it doesn’t, particularly when the signal being updated is used somewhere in another assignment.

Verilog code can be divided in two categories: Synthesizable and Non-synthesizable. The Verilog codes that can be implemented on hardware are called synthesizable. Non-synthesizable codes are those codes which cannot be implemented on hardware. Such codes are useful for testing the designs in simulator and also when the behavior of the design is to be tested without implementing it.

2.3 Verilator

Verilator [11] is an open source tool that can convert synthesizable Verilog code to C++ code. It is also the fastest free publically available Verilog HDL simulator. The converted C++

```
always @
(posedge clock
or posedge reset)
   begin : process_1
   if (reset ==  1'b 1)
      begin
      outp <= 1'b 0;
      end
   else
      begin
      outp <= 1'b 1;
      end
   end
endmodule
```

```
void Vtop::_sequent__TOP__1
(Vtop__Syms* __restrict vlSymsp)
{
    Vtop* __restrict
    vlTOPp VL_ATTR_UNUSED = vlSymsp->TOPp;
    // Body
    // ALWAYS at top.v:45
    if (vlTOPp->reset)
    {
        ++(vlSymsp->__Vcoverage[0]);
        vlTOPp->outp = 0;
    }
    else
    {
        ++(vlSymsp->__Vcoverage[1]);
        vlTOPp->outp = 1;
    }
}
```

Figure 2.2: Verilog Code and its equivalent code in C++ converted code by Verilator

code is referred to as “verilated C++” henceforth. Verilated C++ code represents the same information as present passed in the Verilog code. One example of Verilog code and its corresponding verilated C++ code is shown in Figure 2.2. Verilator also does optimizations in the verilated C++ code to make it run faster. Verilator gives option of instrumenting the code with line coverage, toggle coverage and signal coverage. It also provides a lot of other options to do more analysis and optimizations with the code. This research work uses line coverage option, which adds a line of code to each basic block of the code. Basic block refers to a group of statements which have only one entry and one exit point in a program. The added line follows the format `++coverage[<Specific Number for each basic block>];` The `coverage` array is initialized with zero. If the execution reaches a particular basic block then the value in `coverage`

array for that index is non-zero. This is used to check which basic blocks were reached during the execution.

In this thesis basic blocks are also referred as branches. Figure 2.3 gives example of what is meant by basic block or branch of a code. In the figure, start of each branch is marked with a comment on the first line of the branch. The number associated with each branch is the index of the coverage array instrumented in that branch. This number is referred as coverage number or branch number in this thesis.

2.4 Activating and preceding conditions

Verilated C++ code has multiple, nested if-else blocks. If a particular branch is to be reached in a nested if-else structure then that branch is set as the target branch. The condition just above the target branch is called the activating condition. All the conditions which should be satisfied to reach the activating condition are called preceding conditions.

```
1  if (y > 3)
2  {    ++cov[9];//branch 9
3       y = 7;
4       if (x > 5)
5       {    ++cov[18];//branch 18
6            z = z + 1;
7            x = y;
8       }
9  }
```

Figure 2.3: Sample Verilated C++ code explaining branch numbers and different conditions

For example, in Figure 2.3, if target branch is `branch 18` then the activating condition is `(x > 5)` while the preceding condition is `(y > 3)`. If there were more conditions in the code then there could have been more preceding conditions. By this definition it is obvious that there

can be only one activating condition for a particular branch while the number of preceding conditions can range from zero to any positive integer. Also the preceding and activating conditions will vary based on which branch is set at the target branch.

2.5 Satisfiability Modulo Theory

Satisfiability Modulo Theory (SMT) can be used to check the satisfiability of logical formulae where the domain of the variables can involve integers, real, bit-vectors, etc. For example, if the clauses are `a > 10; a < 15;` and `a + b == 12;` and it needs to be checked if all of them can hold true together, then a SMT solver can be used to get the answer. To do so, an SMT solver instance needs to be created. Each clause of the formula is added as a constraint in the solver instance and the solver solves it to determine its satisfiability. If the instance is satisfiable, then optionally, a model can be generated by the solver listing one possible value for each variable in the instance. SMT has been used for test generation in [12, 13]

It should be noted that only Boolean clauses can be evaluated by the SMT instance, i.e., only the clauses which can be evaluated to either true or false can be added to the instance. Therefore, an assignment like `x = y,` it must be converted to a clause as `x == y` before adding it to the solver. This conversion doesn't modify the fundamental model implied by the assignment. However, if a statement updates a signal using the signal's own value a conflict occurs. In such a case, direct replacement of `'='` with `'=='` would always result in unsatisfiablity. For example, if the assignment is `x = x + 1`, then updating it to `x == x + 1` would always make it unsatisfiable as the same value can't be equal to itself and itself plus one. To handle such cases, Static Single Assignment (SSA) (explained in section 2.7) is used.

SMT draws on the most prolific problems of the symbolic logic: the decision diagram, completeness and incompleteness of logical theories and complexity theory. The computation

complexity of most SMT problems is very high. It is infeasible to build a procedure that can solve arbitrary SMT problems. Therefore, most procedures focus on the more realistic goal of efficiently solving problems that occur in practice. In recent years, there has been an enormous progress in the scale of problems that can be solved due to innovations in core algorithms, data structures, heuristics and carefully considering the implementation details [14]. The annual competition for SAT and SMT procedures is a key ingredient in driving this progress [15].

2.5.1 Z3

Z3 [16] is an open source SMT solver published by Microsoft research. It has been written in C++ and python. The python version of z3 is the SMT solver used in this research. Clauses in Z3 need to be passed in a special format for it to be understood by z3.

For example: Suppose the clause "`(1 & (stato))` be false" is to be added on the z3 solver instance. For it the signal `stato` needs to be initialized in z3. One way of initializing it is

```
z3sigs['stato'] = BitVec(stato, <number of bits in stato >)
```

where `z3sigs['stato']` holds the z3 initialized instance of the variable `'stato'`. After the initialization, the condition would be added in the solver as

`'solver.add((1 & z3sigs[stato]) == 0)'` Note that an extra equality operator and value 0 is added which is not present in the original statement. This addition of details is specific to the kind of clause being added to the solver.

It can be seen that to add any condition from the verilated C++ code to z3 solver, the condition must be transformed to some other format suitable to be read by z3. This transformation is discussed in next section.

2.6 Source to Source transformation

Source to source transformation refers to taking a source code in one language and transforming it to equivalent code in another language. Source to source transformation is helpful in this research to transform verilated C++ code into clauses which can be added on z3 solver.

One way to do source to source transformation is by first extracting the code to be transformed, getting the Abstract Syntax Tree (AST) of it, converting that AST using rules of the other language. The details are discussed as follows

2.6.1 Abstract Syntax Tree

In computer science, an Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of the source code written in a programming language. AST data structure is widely used in compilers [17].

Programming language compilers convert the source code into an AST after parsing it because AST offers many advantages such as

1) It strips off inessential punctuations and delimiters like semicolons, comments etc.
2) It is a tokenized representation of the source code.

These qualities make it an intermediate representation of the source code which can be well understood by compilers. It is also a representation which can be used to generate an equivalent source code in another language.

2.6.2 Pycparser

Pycparser [18] is a parser for the C language, written in pure python. It is a module designed to be easily integrated into applications that need to parse C source code. As it's written in a high level language like python, it's easy to experiment and tweak. It is an open source code

available in new BSD license [19]. It provides method `‘parse’` to generate AST from a C code. There is another method `‘generator’` which generates a C code if an AST is given to it.

For our work the `‘generator’` method was modified to output the AST into a syntax which can be passed to a z3 solver.

2.7 Static Single Assignment

Static Single Assignment (SSA) [20, 21] is primarily used in compiler design for code simplification and optimization. It is a concept which states that all variables should be assigned just once in the whole program. To handle this concept on the source code of real programs, many versions of the variable is used. Once a particular version of the variable is assigned some value, that version of the variable is used in subsequent calls of the variable till the variable is updated again. If any update happens on the variable, then a newer version of the variable is made which is used until it is updated again. For example: There is a code section as

```
x = x + 1;
y = x;
x = y + x;
z = x;
```

After applying SSA the assignments would be updated as

```
x_1 = x + 1;
y = x_1;
x_2 = y + x_1;
z = x_2;
```

where x_1, x_2 are versions of the variable x having the same properties as x. It can be seen here x_1 is assigned value in first statement and it is used in the second and third statements. The third statement tries to update x using the value of x itself, so a newer version, x_2 is made which is used in the fourth statement. This way it is clear that the value of x used in first statement is the value which was passed to the program while the value of x used in second statement was the value updated in first statement.

After application of SSA the flow of values becomes very clear. It allows us to convert assignment statement into equality constraint which is sometimes required for adding constraints to an SMT solver instance.

3 Signal Domain based Reachability Analysis in RTL circuits

Sharad Bagri, Kelson Gent, Dr. Michael Hsiao
"Signal Domain Based Reachability Analysis in RTL Circuits," Proceedings of International Symposium on Quality Electronic Design (ISQED), March 2015, used with permission, 2015

3.1 Overview

Register-transfer level (RTL) verification is a challenging problem for today's complex circuits. A sub-problem of verification is reachability of basic blocks or branches in the code. The work in this chapter proposes a novel analysis based on the domain of signal values in the RTL code to reason about the reachability of all branches without explicit circuit unrolling. This analysis takes into account all assignments, activating and preceding conditions in the code, to derive an assignment table that lists all the possible sequences of branches required to reach a target branch. In the process, it proves unreachable branches as well as provides guidance for reachable branches. This analysis resolves branches more efficiently compared to other methods for the ITC99 [22] benchmarks, especially for larger benchmarks containing previously unresolved branches.

3.2 Introduction

Design verification ascertains that the design meets the specification. Functional validation at the RTL ensures that the RTL design complies with its high level specification. It is estimated that verification consumes at least 50% of the design time [3]. In addition, the efforts required for verification have been increasing at a greater pace than the efforts needed for design [3], making new breakthroughs in verification much more urgent.

Over the years, there has been much research in generating effective stimuli for functional validation. These methods fall in the category of either simulation or deterministic methods. Simulation-based methods have included works based on evolutionary algorithm [23-25], cultural algorithms [26], and ant colony optimization [9]. Examples of recent deterministic methods are STAR [10], HYBRO [8]. There have also been works that combine simulation and deterministic techniques, such as extension of BEACON [27] and [28]. Among all the aforementioned works, determining the reachability of corner cases has been noted as very difficult.

To address these challenges, we propose a new signal domain analysis, performed at the RTL. Our contributions are as follows:

- The reachable values for the relevant signals are derived irrespective of which time-frames that these values can be achieved, reducing computational cost.
- We are able to quickly prove unreachability for many branches.
- If a branch is reachable, we determine which branches are necessary to reach before the target branch. This can be used as a guidance method for test stimuli generation frameworks.

The rest of the chapter is organized as follows. Motivation and related work of this research work are discussed in Section 3.3. Section 3.4 discusses methodology, explained with a working

example, algorithm and program specific details. Section 3.5 discusses how some special cases were handled in the program. Section 3.6 discusses the applications of this work. Section 3.7 compares this method with other similar methods.. Section 3.8 presents the results and analysis of the results. Section 3.9 discusses more about the details of branches which have not been resolved after the application of this work. Section 3.10 concludes the paper.

3.3 Motivation and Related Work

In this section, we will discuss why unreachability arises in circuits and how is it handled in different techniques. We will also discuss about the details which form the background of this chapter's research work.

RTL circuits are described in Hardware Description Languages (HDLs) such as Verilog and VHDL. The code written in HDL is similar to those codes written in High Level Languages (HLLs) such as C++ and Java. HDLs and HLLs have many constructs in common. One such construct is the 'switch' statement. It is a selection control mechanism used when one among many branches are to be selected based on the value of a variable referred henceforth as 'select variable'. The 'switch' construct provides a 'default' case which is a branch taken when the select variable's value cannot match value for entry to any other branch. The 'switch' statement is analogous to the multiplexer in hardware where one among many input lines is selected to pass to the output, based on the value of the select line.

'Default' cases can be used for safety in a design so that behavior can be specified for any unhandled value. Sometimes, a default case may be used to handle branches associated with multiple values. Some hardware designs use a 'default' case as a method of error-checking. In such cases, if the 'select variable' takes an illegal value, then it will be gracefully handled in the 'default' case. If the 'default' case is used for the error-checking then that branch can't be

reached during the legal circuit operation. Hence, the ‘default’ branch becomes unreachable. However, ‘default’ branches may be reachable, for example, when it is part of the design to handle the remaining cases in an incompletely specified ‘switch’ statement. As a result, some default cases may be unreachable in HDL descriptions but this property cannot be generalized.

Unreachability can also stem from state values which may not be achieved during circuit operation. This can either be due to designer intent or due to lack of optimization in the design. If unreachability of an RTL block can be discovered early in the design phase, it can also point to potential design mistakes.

While some hard-to-reach branches have been reached by previous techniques, a number of corner cases remain unresolved. Reachability analysis at the gate level has been proposed in [29]. However, analyses of many of the paths were aborted for complex circuits such as b12 [22]. For unreachable branches, simulation-based techniques can spend considerable effort in an attempt to reach them but ultimately fail. For example, BEACON [9], based on ant-colony optimization, uses pheromones deposited by ants as a measure of the difficulty to reach a branch. . Easy-to-reach branches will have more pheromones deposited because more ants travel these paths more frequently. Later, the easy-to-reach branches are trimmed from the search space so that efforts are concentrated towards hard-to-reach branches. However, as some hard-to-reach branch may be nested inside an easy-to-reach branch, some of these hard-to-reach branches may be missed due to trimming. Our method can help to identify these nested conditions and prevent trimming of branches if the branch contains a nested hard-to-reach branch.

There are methods without explicit circuit unrolling, notable among them is deducing reachable states based on the circuit’s transition relations but it is computationally expensive. In

particular, this method suffers from state space explosion, since there may be an intractable number of states and transitions.

3.4 Methodology

The high level procedure of our approach is given as follows. First, the RTL code is instrumented for line coverage, giving each basic block a unique identifier. Next, all signal assignments are extracted from each basic block. The extracted assignments are grouped based on which signal they modify and labeled with their block identifier. Next, the activating and preceding conditions necessary for reaching a basic block are extracted. For each signal in the activating condition, we add the associated assignments to an 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. Each path in the graph is then formulated as an SMT expression. This expression is solved to determine if the assignments on the path can activate the condition. The basic block identifiers associated with viable assignments are saved. This procedure is repeated for each path in the graph to determine its reachability. This whole procedure is repeated for each condition in the RTL code to find the reachability of each basic block. The reachability can be proved only if the assignment is acyclic, i.e., there is no assignment where the signal updates its own value.

3.4.1 A Working example

```
1  if (y > 3)
2  {    ++cov[9];//branch 9
3       y = 7;
4       if (x > 5)
5       {    ++cov[18];//branch 18
6            z = z + 1;
7            x = y;
8       }
9       else
10      {    ++cov[5];//branch 5
11           x = z;
12      }
13 }
14 else
15 {    ++cov[12]; //branch 12
16      x = 4;
17 }
```

Figure 3.1: Example Verilated C++ code showing only important part of the program

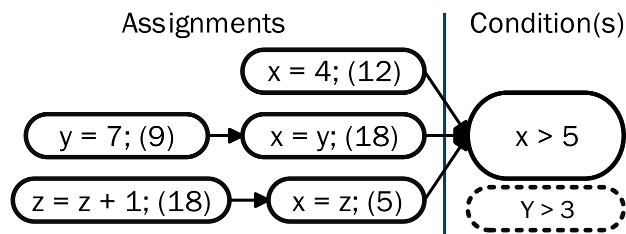

Figure 3.2: Assignment graph for the example code above

Figure 3.1 shows an example C++ code made to explain our method. In this example, variable declarations have been removed and variable names have been shortened to save space while preserving the meaning implied by the assignment. The assignment graph to find reachability of branch 18 for this code is shown in Figure 3.2. In order to reach branch 18, the target condition is '`(x > 5)`'. Here, the activating condition is '`(x > 5)`', and the preceding

condition is '`(y > 3)`' shown by the dotted oval on the right side of Figure 3.2. Since the activating condition '`(x > 5)`', involves '`x`', signal '`x`' becomes the target signal. We initially assume that all assignment statements are reachable without any constraints. All assignment statements to signal '`x`' are extracted that are '`x = 4`', '`x = y`', and '`x = z`'. These form the first level of the graph. The second assignment of '`x`' has a new signal '`y`' while the third assignment has a new signal '`z`'. For the first assignment the path terminates with the assignment of '`x = 4`' because there is no new signal involved. For the second assignment, all assignments of '`y`' are added to the path to form the second level of graph. Also the preceding condition '`(y > 3)`' is added to the path as it also involves the new signal '`y`'. For the third assignment, the assignment '`z = z + 1`' is added. As there are no preceding conditions involving '`z`' so no condition except the activating condition is added while solving for this path. The conjunctive formulae, created for each path is as follows:

1. `(x > 5) ∧ (x == 4)`
2. `(x > 5) ∧ (x == y) ∧ (y > 3) ∧ (y == 7)`
3. `(x > 5) ∧ (x == z) ∧ (z == z_1 + 1). (Note that SSA is first applied before adding to the instance.)`

Each path would be solved independently. It can be seen here that formulae 2 and 3 are satisfiable while formula 1 is not. So, the basic block numbers for satisfiable assignments are ((18), (9)) and ((5), (18)). Similarly, the unsatisfiable assignment is (12). It means that the assignment in block number 12, i.e., '`x = 4`' won't satisfy the target condition '`(x > 5)`', but the other assignment might.

The assignment '`z = z + 1`' is cyclic, as the signal updates its own value. Therefore, its reachability can't be proved. . This is because we do not know the starting value of '`z`' and how many times this assignment may be called, which depends on its placement in the code and

number of cycles for which the verilated C++ code is simulated. Without this knowledge the set of reachable value of '`z`' becomes the set of values possible within its bit-width limit. This value would propagate to '`x`' making the target branch '`(x > 5)`' satisfiable with this assignment.

Note that we are not taking any loops in consideration in our analysis because synthesizable HDL does not allow variable-length loops, where the number of iterations is resolved at runtime. Also, for the loops which are present, the number of iterations is known at compile time and the loop can be unrolled during synthesis. We have only considered assignments, activating and preceding conditions. Therefore, there is no chance to have an intractable number of paths for loops.

The output of our analysis is an assignment table which lists the branch number combination(s) required to reach each branch. A sample of an assignment table is as follows:

84 S ((38, 98, 101)), ((36, 82, 83), (68))

This means in order to reach target branch of 84, the assignments in any of the branches of (38, 98, 101) must be executed. The branch number separated by comma specifies that all the three branches have the same assignment on the target variable and so they are grouped together. Another solution is (36, 82, 83), (68) which means any branch, 36, 82 or 83, should be executed before the target branch and branch 68 should be executed before execution of any of the branch of (36, 82, 83).

With this example in mind, the next section discusses the algorithm of our method.

3.4.2 Algorithm

Algorithm 1 shows the pseudo code of the main function of our method. All the branches of Verilated C++ code are referred by their branch numbers. While reading in the Verilated C++,

Algorithm 1: Main Function Algorithm

```
1: Initialize and read in verilated C++
2: for each signal do
3:     group its assignment along with block number
4: end for
5: for each branch of verilated C++ do
6:     Extract its preceding and activating conditions
7:     Extract signal(s) from the activating condition. It is 'sig_on_decision'
8:     Add activating and relevant preceding condition on z3 solver
9:     Incrementally solve the instance by calling recursive function for
       sig_on_decision
10: end for
11: Print results
```

all the signal names and the input signal names are extracted. All assignments are clustered based on which signal is being assigned which is shown line 2 and 3 of the algorithm. If the same assignment is present in multiple blocks, they are all grouped together. Next, we go to each target branch whose reachability is to be determined. The activating condition is extracted for the target branch. The signal(s) involved in the activating condition is extracted and is the signal(s) with which the list ‘sig_on_decision’ is started. A recursive function (described in Algorithm 2) is called for signals in ‘sig_on_decision’. If the activating condition is an ‘if’ construct, the condition inside it must be true. Otherwise, the condition is the ‘else’ branch and the corresponding ‘if’ condition needs to be falsified. Verilated C++ doesn’t have ‘else if’ because it implements it as an ‘if’ condition inside an ‘else’ block. The signal to be added from ‘sig_on_decision’ is referred as ‘sig_to_add’ in the recursive function.

Algorithm 2 shows the high level overview of the recursive function. This function's primary purpose is to add one new signal to the solver instance at a time. This function is initially called from the main function once for each target branch. Later, it may call itself recursively with new signals detected in any of the assignments. If the 'sig_to_add' is found as

Algorithm 2: Recursive algorithm

```
   Input: sig_to_add - New signal to be added in solver
 1 if sig_to_add is Input then
 2 |  path is satisfiable, return
 3 if recursion depth > Max allowed depth then
 4 |  solve() and return
 5 if any preceding condition contains sig_to_add then
 6 |  add the condition to solver
 7 for each assignment of sig_to_add do
 8 |  apply SSA
 9 |  add the assignment to solver and solve()
10 |  if satisfiable and new signal present in assignment then
11 |  |  call recursion with new signal
12 |  remove the assignment from solver
```

an input signal of the RTL circuit, then the corresponding path is returned as satisfiable. It is because input is always fully controllable and can be given any value within its bit-width limit. Next the recursion depth is checked. Recursion depth is specified by the user at the beginning of the program. If the recursion depth is reached, the instance is solved and the result is returned. Next, if any preceding condition(s) is found which involves signal(s) present in 'sig_on_decision', it is added to the solver instance as additional constraint.

Next, each assignment of the 'sig_to_add' is considered separately. First, if needed, SSA is applied to the assignment. Next, if the signals in the assignment are of different bitvector length then, necessary extensions of the bitvectors are done, to match lengths, to ensure a valid

SMT expressions. The assignment is added to the solver and checked if the path until the call of this instance of recursion is satisfiable. If it is satisfiable and no new signals are involved in this assignment, then the path is added to the list of satisfiable path. However, if any new signal(s) is involved then the recursion is called with the new signal. Once a decision has been done on the assignment under consideration, the analysis moves on to the next assignment of the 'sig_to_add' until all the assignments of 'sig_to_add' have been analyzed.

3.4.3 Program Input

The full program has been written in Python and requires formatted C++ code converted from Verilog to C++ by Verilator. The converted code is formatted in 'allman' style of 'Astyle' [30]. Following these formatting, all broken lines are brought together so that they become one statement per line. Example: verilated C++ may have code as:

```
1.      vlTOPp->text_out[3] = ((0xffffff00 & vlTOPp->text_out[3])
2.                            | (0xff & ((IData)(vlSymsp->TOP__v__DOT__us33.d)
3.                                       ^ vlTOPp->v__DOT__w0)));
```

Successive lines are brought together until that line becomes a complete statement as shown below:

```
1.      vlTOPp->text_out[3] = ((0xffffff00 & vlTOPp->text_out[3]) | (0xff & ((IData)(vlSymsp->TOP__v__DOT__us33.d) ^ vlTOPp->v__DOT__w0)));
```

3.4.4 Program Output

The program prints the output in two formats. One is a detailed format which lists how each of the conclusions of satisfiable or unsatisfiable was reached. This is human readable and helps to understand how each conclusion was derived. Other is a short format which prints all details in a file in a concise manner which can be easily parsed by another program. The output of this algorithm is referred as assignment table.

3.4.4.1 Human Readable Format

The human readable format is broadly of three types. The three types are based on the characteristics of signals and the assignments involved.

1) When signals in conditions are inputs
2) When signals in conditions are assigned constant values
3) When signals in conditions are assigned values which are again signals.

Case 1) when signals on the conditions are inputs:

```
1.      Coverage No. to test : 2
2.      Coverage 2 requires ((line1) & (line2)) to be false
3.      Conditions involving Sig On Decision :
4.      ((line1) & (line2)) to be false
5.      It is an input so can take any value, one solution is
6.      ['I']
7.      [line1 = 0]
```

Figure 3.3: Detailed output when Signals in Conditions are input

If the recursive function determines that all signals in sig_on_decision are inputs, then there are no constraints on the values each signal can take. Thus, z3 solver is called to find one solution which would satisfy the constraint. The solution is printed as a supporting example for branch reachability.

Case 2) Signals on conditions are only assigned constant values

```
1.      Coverage No. to test : 3
2.      Coverage 3 requires (0 == (v__DOT__process_1_stato)) to be true
3.      Conditions involving Sig On Decision :
4.      (0 == (v__DOT__process_1_stato)) to be true
5.      (((((((0 == (v__DOT__process_1_stato)) || (3 == (v__DOT__process_1_stato))) || (1 == (v__DOT__process_1_stato))) || (4 == (v__DOT__process_1_stato))) || (2 == (v__DOT__process_1_stato))) || (5 == (v__DOT__process_1_stato))) || (6 == (v__DOT__process_1_stato))) || (7 == (v__DOT__process_1_stato))) to be true
6.      Unsatisfiable Solution is :
7.      v__DOT__process_1_stato == 5 : [7, 10]
8.      Unsatisfiable Solution is :
```

```
9.      v__DOT__process_1_stato == 6 : [14, 17]
10.     Unsatisfiable Solution is :
11.     v__DOT__process_1_stato == 7 : [13, 16]
12.     Satisfiable solution is :
13.     v__DOT__process_1_stato == 0 : [0, 20, 23]
14.     Unsatisfiable Solution is :
15.     v__DOT__process_1_stato == 1 : [2, 5]
```

Figure 3.4: Detailed output when Signals in Conditions are assigned constant values

When the signals in the condition are not primary inputs, then, verilated RTL function is searched for all assignment to signals in sig_on_decision. It is possible that the target signal may be assigned the same value at multiple places in the code. In this case, all coverage numbers are saved. The details in Figure 3.4 are:

- Line 1 lists the coverage number which is the target coverage
- Line 2 lists the activating condition for the target coverage. This condition is just above the code block where the target coverage exists. The signal(s) inside this condition is sig_on_decision
- Line 4, 5 lists all conditions where sig_on_decision is involved.

These conditions are pushed on solver. Then, each assignment of signals in sig_on_decision is checked for its Satisfiability. It is done by pushing the z3 equivalent clause of the assignment and solving it. After the solution, the clause is popped from the z3 context and next one is pushed.

- Line 6 onwards, each assignment's solution is displayed along with the assignment itself. The numbers in square bracket next to the assignment give the coverage identifier for the assignment. If there are multiple code blocks in which equivalent assignment exists, then all respective coverage numbers are separated by commas.

  In this case, line 6 and 7 give an example of an unsatisfiable solution.

Similarly, line 12 and 13 list a satisfiable solution.

It can be checked that for conditions listed in line 4 and 5, the assignment of line 7 is unsatisfiable while assignment of line 13 is satisfiable.

Case 3) when signals in conditions are assigned values which are other signals

```
1.      Coverage No. to test : 84
2.      Coverage 84 requires (0 == (v__DOT__count2)) to be false
3.      Conditions involving Sig On Decision :
4.      (0 == (v__DOT__count2)) to be false
5.      Unsatisfiable Solution is :
6.      v__DOT__count2 == 0 : [28]
7.      Satisfiable solution is :
8.      v__DOT__count2 == 63 & v__DOT__count2_1 - 1 : [38, 98, 101]
9.      Satisfiable solution is :
10.     v__DOT__count2 == 33 : [45]
11.     Satisfiable solution is :
12.     v__DOT__count2 == 8 : [69, 87, 91, 94, 97, 100]
13.     Unsatisfiable Solution is :
14.     v__DOT__count2 == v__DOT__timebase : [36, 82, 83]
15.     v__DOT__timebase == 0 : [28]
16.     Satisfiable solution is :
17.     v__DOT__count2 == v__DOT__timebase : [36, 82, 83]
18.     v__DOT__timebase == 63 & v__DOT__timebase_1 - 1 : [68]
19.     Satisfiable solution is :
20.     v__DOT__count2 == v__DOT__timebase : [36, 82, 83]
21.     v__DOT__timebase == 33 : [32]
```

Figure 3.5: Detailed Output when signals in conditions are assigned values which are signals

In this case, lines 1 to 12 are the same as in the previous case.

Line 13 - 15 presents a case in which a new signal is found during the evaluation of an assignment. Line 14 finds an assignment which is dependent on another signal. So, the assignment of that new signal is searched in the verilated RTL function and they are solved together to see if it is satisfiable or not.

Line 18 presents a case where SSA has been applied on the signal as the signal on both right and left hand side are same. Application of SSA in our case has been discussed in more detail in section 3.5.1

3.4.4.2 Short Format or computer parse-able format:

The short format is a condensed version of human readable format. This file is intended as a method for passing information from the algorithm to other programs. This would help to have the other program with a simpler parsing grammar than the long format. In this format, each line lists all satisfiable and unsatisfiable coverage numbers for each target coverage number.

The first number in each line is the coverage number associated with the target condition. It is always followed by character 'S' standing for Satisfiable. It is followed by coverage numbers which represents the coverage number of the assignments that satisfy reaching to target coverage. After that, character 'U' is printed which stands for unsatisfiable. It is followed by coverages of the assignments which don't satisfy the condition of target coverage. A sample output is printed in Figure 3.6

```
1.      2 S ['I'], U
2.      3 S [[0, 20, 23]], U [[7, 10]], [[14, 17]], [[13, 16]], [[2, 5]],
3.      84 S [[38, 98, 101]], [[45]], [[36, 82, 83], [68]],
4.      84 S [[38, 98, 101]], [[45]], [[36, 82, 83], [68]], [[36, 82, 83], [32]],
   U [[28]], [[36, 82, 83], [28]],
```

Figure 3.6: Output of short format

The details of Figure 3.6 are given as follows: In line 1, the first number 2 stands for target coverage. It is followed by 'S'. ['I'] after S indicates that the signal involved in this condition is an input so it is always satisfiable. No coverage is printed after 'U' as it is always satisfiable.

In line 2, the signal in the condition for target coverage point 3 is not an input. In this case, it lists all the satisfiable coverages after 'S' and unsatisfiable ones after 'U'. All coverage numbers (separated by commas) stand for coverages which have the same assignment. For example, in line 2, coverage 0, 20, 23 have a statement as `' v__DOT__process_1_stato = 0'`

which satisfies reaching the target condition of `‘(0 == (v__DOT__process_1_stato)) should be true’` of coverage number 3.

Line 3 lists the details for coverage 84. Here the third set of satisfiable solution is listed as `[[36, 82, 83], [68]]` which means that first, the assignment covered by `[68]` should be executed followed by one the assignment contained in `[36, 82, 83].`

3.5 Some Special Cases

Some specific constructs in the Verilated C++ code has been handled specially in our method. We discuss about them in this section.

3.5.1 Handling of Static Single Assignment

As an SMT solver only takes Boolean clauses as predicates, the normal way to enter an assignment in the solver is to replace ‘=’ by ‘==’. In some cases same signal may appear in both the right and left sides of assignment. For example

```
1.      'v__DOT__counter = (7 & ((1) + (v__DOT__counter)));'
```

In such cases replacing assignment by equality and putting it as a clause in SMT solver would always return the clause as unsatisfiable since x is always unequal to (7 & (1 + x)). However, in reality it will not be the case. In reality, the signal on right and left hand sides would have different values. In such case, the right hand side of signal is replaced by signal suffixed with ‘_1’ or ‘_(Signal Counter Value)’. The new expression for the above example would be

```
1.      'v__DOT__counter = (7 & ((1) + (v__DOT__counter_1)));'
```

The full Algorithm to apply SSA for each level of recursion is given in Algorithm 3.

Algorithm 3: SSA update algorithm

```
   Input: ssa_count - Counter of SSA signals
   Input: assign - Assignment on which SSA is to be applied
1  Extract signals on left and right side of assign
2  for each left side signal do
3      if ssa_count is not zero then
4          update the signal as signal_(count_value)
5  for each right side signal do
6      if signal counter initialized in ssa_count then
7          Update counter value by 1
8          Update the signal name with signal_(count_value)
9      else
10         Initialize the counter with 0
11 if any update happened then
12     Update the assign with new signals from line 4 and 8
13     return the new signals, updated assignment from line 12
14 Else return None
```

3.5.2 Non-blocking assignment handling in verilated RTL function and our code

For a non-blocking assignment in Verilog code, Verilator converts it to C++ code in the following way.

A copy of the signal is made at the beginning of verilated RTL function. All decisions are made on the original signal while any update of the signal happens in the copy. At the end of verilated RTL function, the copy's value is copied to the original signal. Thus the update happens at the end of the cycle which is equivalent to update happening at the beginning of next cycle.

An example of such a kind of assignment shown in Figure 3.7

```
1.      void Vtop::_sequent__TOP__1(Vtop__Syms* __restrict vlSymsp)
2.      {
3.      __Vdly__v__DOT__data_out = vlTOPp->v__DOT__data_out;
4.      (All branches of code)
5.      vlTOPp->v__DOT__data_out = __Vdly__v__DOT__data_out;
6.      }
```

Figure 3.7: Non-blocking assignment code in verilated RTL function

Here actual signal is `vlTOPp->v__DOT__data_out` while `__Vdly__v__DOT__data_out` is the copy of it.

As these assignments happen both at the beginning and at the end of the function; they are part of the main stem of code and not part of a branch in the code. These assignments would always be executed when verilated RTL function is called. Verilator doesn't assign any coverage number to the main stem of code. However, in our code for generating the assignment table, each statement is associated with a coverage number. Thus, the assignment statements in the main stem of the code are assigned coverage number of -1 to make it consistent with how other statements are treated in our code.

If this combination of assignments is seen by the code generating assignment table, i.e., lines 3 and 5 of Figure 3.7, it is discarded as after application of Single Static Assignment, the signal can take any value. However, in most cases, some other branch of code would modify the value in copy in between and this combination won't be meaningful. If there is no modification in between, then the same value would be copied to next cycle and so there would be no modification in the variable's value. In both cases, in real circuit the signal can't take just any value but the assignment table would say so.

3.5.3 Assignments from array

An assignment from an array, where the array index is also a variable, is ignored and is treated as a free variable because there are several possible values for such an assignment. For example, in the assignment `data_out = memory[address];` there are two variables to keep track when assigning a value to the left side signal `'data_out'`, namely `'memory'` and `'address'`. We will have to track all places in the code where any write is happening to `'memory'` array as well as any write happening to the `'address'` variable. The presence of these

signals together will require us to keep track of potentially large memory arrays. To stop potential state space explosion as well as reduce computational cost and complexity, we treat the right side as an unconstrained variable of the same type as `‘data_out’`.

3.5.4 Assignment of different bitvector length where left hand side signal has higher bits than right hand side signal

Verilog specifies precisely how many bits are needed for each signal. The number of bits in a signal in Verilog can be any positive integer. It is unlike in other programming languages where signals are byte aligned i.e. they are multiples of 8 bits.

Bitvectors are array of bits. While declaring them, the specific number of bits the bitvector needs must be specified. Hence in our case all signals are created as bitvector in the SMT solver.

However, if an assignment has signals of different bitvector lengths, then the SMT solver can’t solve it. To handle such a case the bitvector is extended by putting zeroes on the most significant bits until signal on right hand side of assignment becomes equal to signal on left hand side. The vice versa case will be discussed in Section 3.5.5

3.5.5 Assignment of signals with higher bitvector length on right hand side than left hand side signal

If the verilated RTL function has an assignment where the signal(s) on the right hand side has higher bitvector length than the left hand side signal, such an assignment is ignored. It is because the bitvectors need to be extended on the right hand side. Such an extension can be done only if a modulus is involved on right hand side signal, otherwise it would lead to erroneous values. If a modulus is present on right hand side of the code then left side signal can be

extended by zeroes on the most significant bits, but it would add a lot of unnecessary complexity in the code without adding much value. With this program, we are interested in only getting a heuristic which can prove unreachable branches easily, so doing expensive computations may be overkill at this point and not be worth it.

3.5.6 Assignment involving more than two signals of different bitvector lengths.

If there is an assignment which has more than two signals of different bitvector lengths like `Sig_bv_len_10 = Sig_bv_len_5 + Sig_bv_len_4`. Such an assignment is not bit extended to save the complexity involved in doing it.

3.5.7 Depth of Analysis

This analysis can be done recursively so this analysis can be done till any level of depth. However we need to keep few things in mind before going for higher depth.

3.5.7.1 Finding default cases

All default cases of Verilog code (as in a switch statement) can be found in a maximum of two levels. If signals have non-blocking assignments, they require 2-level deep analysis to find the default case. The details of implementation of non-blocking assignment is explained in Section 3.5.2. If a signal is a blocking default case, then it can be found out in just one level of analysis. (Blocking and non-blocking signals would be explained in background)

3.5.7.2 Prevent explosion of search space

As we go deeper in levels of analysis, the number of signals in sig_on_decision keeps on growing and eventually it will involve all the variables of the program. When more variables are

involved in sig_on_decision, more and more coverages are added to the satisfiable coverages. There is a point where adding more variables will not provide any additional value. It was found that if program has few variables then the analysis reaches saturation faster, i.e. after level 2 or so, the analysis returns same assignment table for any number of levels. If the program has lot of variables which depend on each other, then the analysis at different recursion depths returns different assignment table.

3.6 Application of this work

This work can be used for two purposes. 1) Discover unreachable branches in RTL code and 2) Provide heuristic to test vector generation program

3.6.1 Discovering unreachable branches

The assignment table can help in discovering unreachable branches. Generally, default cases in Verilog code are unreachable branches. In verilated RTL function, the default condition is translated to an unsatisfying trailing else condition in an if-else block. For example, in circuit b06, coverage number 22 `(1 & (v__DOT__curr_state))` is unreachable. The assignment table output for that coverage is shown in Figure 3.8

```
1.      Coverage No. to test : 22
2.      Coverage 22 requires (1 & (v__DOT__curr_state)) to be true
3.      Conditions involving Sig On Decision :
4.      (1 & (v__DOT__curr_state)) to be true
5.      (2 & (v__DOT__curr_state)) to be true
6.      (4 & (v__DOT__curr_state)) to be true
7.      Unsatisfiable Solution is :
8.      v__DOT__curr_state == __Vdly__v__DOT__curr_state : [-1]
9.      __Vdly__v__DOT__curr_state == 2 : [10, 4]
10.     Unsatisfiable Solution is :
11.     v__DOT__curr_state == __Vdly__v__DOT__curr_state : [-1]
12.     __Vdly__v__DOT__curr_state == 5 : [5]
13.     Unsatisfiable Solution is :
14.     v__DOT__curr_state == __Vdly__v__DOT__curr_state : [-1]
```

```
15.     __Vdly__v__DOT__curr_state == 4 : [7, 16]
16.     Unsatisfiable Solution is :
17.     v__DOT__curr_state == __Vdly__v__DOT__curr_state : [-1]
18.     __Vdly__v__DOT__curr_state == 6 : [19, 17]
19.     Unsatisfiable Solution is :
20.     v__DOT__curr_state == __Vdly__v__DOT__curr_state : [-1]
21.     __Vdly__v__DOT__curr_state == 1 : [20, 8, 14, 3]
22.     Unsatisfiable Solution is :
23.     v__DOT__curr_state == __Vdly__v__DOT__curr_state : [-1]
24.     __Vdly__v__DOT__curr_state == 0 : [0, 22]
25.     Unsatisfiable Solution is :
26.     v__DOT__curr_state == __Vdly__v__DOT__curr_state : [-1]
27.     __Vdly__v__DOT__curr_state == 3 : [13, 11]
```

Figure 3.8: A sample output of our method for an unreachable branch

In Figure 3.8, for the branch number 22, sig_on_decision is `'v__DOT__curr_state'.` All constraints on the signal required to reach that coverage are listed in lines 4-6. It can be seen that if value 7 is assigned to sig_on_decision, all those constraints would be satisfied and the target coverage would be reached. `'v__DOT__curr_state'` is assigned a value at only one place with other signal `__Vdly__v__DOT__curr_state` as listed in line 8, 11, 14 etc. Line 9, 12, 15 lists all assignments of `__Vdly__v__DOT__curr_state` and it can be seen that it is nowhere this variable is assigned a value greater than 6. However, looking on all the constraints on that signal listed in line 4-6, it can be seen that it would require a value of 7 to satisfy all the conditions. After line 6 each solution result from the SMT solver is shown. It also lists what particular clauses where pushed in SMT solver for solving it. Hence there is no assignment which satisfies reaching this coverage `(1 & (v__DOT__curr_state))`. Thus, it can be proved that it is unreachable.

3.6.2 Cases of Unreachability

Unreachability can be divided into two categories based on the levels in assignment table. The two cases and their detection are described in the following sections.

Case 1) Only constants are assigned in level 1

When only constants are assigned as values to sig_on_decision and none of such assignments satisfy the constraint in the target condition. For example: If target condition is `a>10` and none of the assignments assign any value greater than 9 to `a`.

Case 2) Only constants are assigned in level 2

When there are some assignments in the first level that depend on other signals, but those signals are only assigned constant values in level 2. Furthermore, none of those assignments in level 2 satisfy the constraint. For example: If the required condition is `a>10` and there is an assignment as `a=b` but `b` is assigned only constant values and has no assignment that assigns it a value greater than 9.

3.6.3 Provide heuristic to test vector generation program

The assignment graph can also aid test generation by providing heuristics to guide the search process. Each satisfiable path in the graph represents a possible way to activate the condition. If the test stimuli generation efforts are concentrated on each of the satisfiable paths then the target branch has an improved chance of being reached.

3.7 Comparison with other methods

While this may resemble cone-of-influence (COI) or bounded cone-of-influence (BCOI) [31] reduction, we do not need to include the cycle/frame information. In other words, all the constraints are considered without explicitly defining the cycle in which they are assigned. The conditional expressions, other than the conditions on the target signal, can also be neglected. In doing so, the instance can be significantly smaller than a Bounded Model Checking (BMC) [32]

instance even with COI or BCOI reduction. The BMC instance may require a long sequence of frames [33-35] while such sequencing requirements are unnecessary in our setup. Furthermore, we extract as much signal value information as possible from the relevant assignment statements to reason about the values of intermediate signals. This allows us to constrain the search with the set of possible values each relevant signal can take. Our analysis is analogous to performing a reachability computation on a program slice for the target branch. However, our abstraction removes many details and adds useful constraints to the instance at hand.

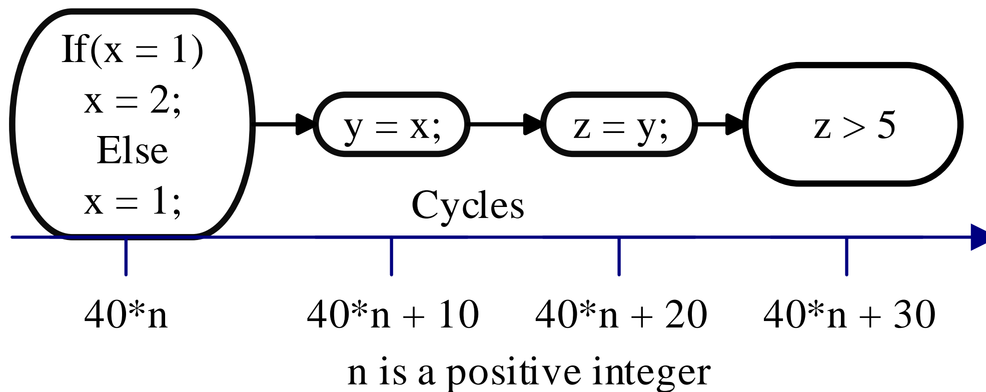

Figure 3.9: An example RTL circuit representation for comparing with our method

Our method returns superior results compared to BMC. Figure 3.10 shows an RTL code representation where BMC will require many cycles to resolve. In this example, the number 10 is selected at random and any other big number selected here would also work fine to explain the concept. The detail of the circuit is as follows. Activating condition of a branch is ‘`(z > 5)`’. ‘`z`’ is updated 10 cycles before as ‘`z = y`’. ‘`y`’ is updated at the preceding 10th cycle as ‘`y = x`’ and ‘`x`’ is updated at preceding 10th cycle as ‘`x = 1`’ or ‘`x = 2`’. In this case, BMC will have to unroll for at least 40 cycles to prove the unreachability while our method would prove the unreachability with just 3 levels of recursion or, in other words, third level of the graph. We show this example in this form rather than the code form because the code for it would be very

long. One example where this kind of condition can appear is in counter circuits where some value may be toggled based on when the bit at a Most Significant Bit (MSB) toggles or some bits which are near MSB.

3.8 Results

The experiments were run on a laptop with 8 GB RAM, Intel® Core™ i5-3210M CPU@2.5 GHz running Windows 7, 64 bit operating system. Code is written in Python version 2.7.5. The Python code was running always on one of the four processors. We perform our analysis on ITC99 [22] benchmark circuits that contain previously unproven reachability points. The results are shown in Table 1.

Table 3.1: Details about resolution of unreachable branches by our method

Circuit	Max Unreachable branches	Proved in [27]	Time (s) [27]	Proved by our method	Time (s) Ours
b06	1	1	0.02	1	2.9
b07	2	1	0.41	1	3.1
b10	1	1	3.12	1	3.2
b11	1	1	4.2	1	3.1
b12	2	1	69.6	2	5.2
b14	17	2	94.2	12	7.2
Or1200-0	2	0	2.82	1	2.9
Or1200-1	1	0	2.75	1	3.2

In Table 3.1, the first column lists the selected circuits. For each circuit, the second column reports the minimum number of branches which have not been resolved by other techniques such as [8-10, 27]. These papers have reported the best branch coverages and so they are used as comparison, as they provide the upper bound on the number of unreachable branches. The third column lists the number of branches which have been proved unreachable by [27] and the fourth column lists the time taken in seconds for the complete algorithm in [27] to run. The separate amount of time required to run BMC has not been reported in the paper. The fifth

column lists the number of branches proved unreachable by our method and the sixth column shows the time taken for running our algorithm, only on the branches which are proved unreachable by our method.

The technique in [27] uses BMC to check reachability of undetected branches during functional stimuli generation. It can be seen that there is a significant improvement in the number of branches proved unreachable by our technique in larger circuits. For example, in b14, 17 branches were not resolved by previous methods. Our method was able to resolve 12 of them in just 7.2 seconds while BMC in [27] was able to resolve only 2. The BMC uses k-induction, as described in [36], with k set to value of 2, i.e., the circuit is unrolled for two cycles. However, without the information on the set of reachable values for some of the signals that the target depends on, some branches may be reachable from an illegal starting state, forcing BMC to falsely conclude that the branch might be reachable. Theoretically, further unrolling will identify more unreachable states but it is computationally infeasible for a large number of time frames. Conventional BMC as the one used in [27] is analogous to performing the analysis in the time domain with explicit unrolling of the circuit, while our approach discards such time-frame constraints.

In terms of execution time, the cost for various levels of recursion to analyze all branches is shown in Table 3.2.

In Table 3.2, the first column lists the circuit name. The total number of branches for each circuit is listed in second column, followed by the time taken for each level of recursion. As we have kept the levels of recursion as a user configurable value, we reported results on different depths of recursion. We wanted to see the effect of level of recursion has in our program.

Typically, for the proven unreachable branches, a maximum of only two levels of recursion were required.

Table 3.2: Run time for different levels of recursion for all branches for our method

Circuit	Number of branches	Run Time (s) for recursion depth (K) of our method			
		K=1	K=2	K=3	K=4
b01	26	8.0	7.9	7.9	7.9
b06	24	4.2	6.1	6.1	6.1
b07	20	4.7	7.9	7.9	7.4
b10	32	6.5	12.7	12.7	12.6
b11	33	12.9	15.2	15.0	15.1
b12	105	60.2	62.2	60.7	60.8
b14	211	63.8	166.9	567.5	1225.3
or1200-0	17	4.1	4.7	4.7	4.6
or1200-1	23	4.8	5.7	5.7	5.7
or1200-2	17	3.6	3.5	3.5	3.5
or1200-3	47	11.5	13.9	13.9	13.9

The execution times in Table 3.1 and Table 3.2 differ because Table 3.2 lists execution times when the analysis is run on all the branches of the circuit. Table 3.1 lists time for only the branches which we were able to prove unreachable by our method. Also we noted that all the unreachable branches were proved unreachable within a recursion depth of 2 of our method. Once a branch is proved unreachable, the run time would not increase even if the number of recursion depth is increased as there would be no new layers of assignments to add in the assignment graph. This is because the fixed point in the path would already have been reached.

In our approach, all required initialization for determining reachability of each of the branches is performed at the beginning to save recurring computational costs. However, it has a large overhead in setup time as it is implemented in Python. This cost becomes insignificant in larger circuits. In addition, the Python version of Z3 is used. The cost may be reduced if the C++ version of Z3 was used. Finally, the CPU used for this experiment was computationally inferior compared to that in [27] which may contribute to the increased computation costs.

3.9 Analysis of unreached coverages

In b07, coverage point #13 is not reached and isn't proved reachable by the assignment table. It looks to be unreachable from a manual analysis. At that point in the code, fixed memory locations are read and their values summed. This value should not be equal to 2 to reach this coverage. However, the sum of them is two. This is currently difficult to do by the assignment table because a) the value is being read from an array, and b) the reading is done at a much earlier point from this coverage number.

In b14, coverages 17, 21, 47, 191, 194 haven't been reached and also haven't been proved unreachable. The technique in [27] is able to prove only 2 branches as unreachable for b14.

From a manual analysis it can be seen that branches 17 and 21 can be reached if a specific input is given in one of the cycles. It is extremely difficult as there is only one input out of 2^{32} possible inputs for that cycle. Also it is not a constant input. It depends on the value of different registers. The values of each of those register keeps on changing in alternate cycles.

Coverage 47 requires specific values on three registers. It is not verifiable if all of them can be reached together.

Coverage 191 requires a value which is not verifiable to be reached or not.

Coverage 194 is unreachable from manual analysis. It requires a value of 4 on a register. That register is being assigned constant values of 0 to 3. At one place it is assigned a value from a register which is of higher bitvector length. Modulus of 4 is being done on that value before being assigned. So manually we know that it will never reach a value of 4. However, proving it is difficult because the bitvector length on the right hand side variable is greater than that of

bitvector length on left hand side. In such case bit extension is not being done as discussed in section 3.5.5. The assignment is like this

```
1.      mf = (3 & ((IR / 0x8000000) % 4))
```

where `mf` is of 2 bits and `IR` is of 32 bits. As modulus is there `mf` can be bit extended by zeroes but if modulus was not there the bit extension on left hand side signal would lead to erroneous values. Hence to preserve the generic nature of the code, this bit extension is not done.

3.10 Conclusions

We have presented a novel method to resolve reachability of each branch of the RTL code. As our technique performs the analysis in the signal value domain, the reachable values for the relevant signals are computed irrespective of which time-frames that these values can be achieved. This formulation drastically reduces the possible number of values of the signal from any unconstrained value to a possibly few set of values. It finds the set of assignment statements and reasons about the values that can be assigned to each signal. Thus, the unreachability of many previously unresolved branches can be resolved. Also, the information learned in the process can aid in the generation of input stimuli for RTL testing by providing guidance to the test pattern generation frameworks, which is part of the future work.

4 Reaching Branches in RTL code with Cycle-By-Cycle Restrictive Symbolic Execution

4.1 Overview

We propose a symbolic execution method restricted in the number of clauses and time frames analyzed. A random or user-defined input sequence is used to generate an initial execution trace. The trace is analyzed to find the clock cycles in which a target branch can potentially be reached. For each of those cycles, the code is flattened and the statements influencing the target branch's activating condition are added as constraints to an SMT solver. If any of the statements involves an input signal, then the values of the remaining signals are taken from previous cycle and added as additional constraints. The solver then returns the values for the circuit's input(s) to reach the target branch. This approach drastically reduces the number of clauses to be analyzed compared with conventional symbolic execution. This restrictive symbolic execution is shown to be much more efficient for hybriding with a simulation-based engine. In addition, for those branches reached by the trace, the method can find a shorter-length

sequence to reach them. Finally, some branches can be proven as unreachable in the search process.

4.2 Introduction

Design verification and validation ascertains that the design meets the specification. Functional validation at the Register Transfer Level (RTL) ensures that the RTL design complies with its high level specification. It is estimated that verification consumes at least 50% of the design time [3]. In addition, the efforts required for verification have been increasing at a faster pace than the efforts needed for design [3], making new breakthroughs in verification much more urgent.

Over the years, much research effort has been focused towards generating effective stimuli for functional validation. These methods fall in the category of either simulation or deterministic methods. Simulation-based methods include works based on genetic algorithm [23, 24], cultural algorithms [26], and Ant colony optimization [9]. Examples of recent deterministic methods include STAR [10], HYBRO [8]. There have also been research works that combine simulation and deterministic techniques, such as extension of BEACON [27]. Among all the aforementioned approaches, determining the reachability of corner cases has been noted to be very difficult.

To address these challenges, we modified the symbolic execution [37] in order to improve its run-time performance while retaining its characteristic benefit of reaching corner cases. Although conventional symbolic execution systematically explores the search space by visiting each branch of RTL code via some strategy, such as the depth-first traversal, the cost can be enormous as there might be many paths that need to be explored. In the proposed method, we first perform simulation with an input sequence, also referred as vector, which can be generated

either randomly or by any other method. We analyze the paths that have been exercised by taking advantage of the power of symbolic execution, but in a restrictive manner to keep the search space small. Our symbolic analysis is restricted to be on a cycle-by-cycle basis. Unlike conventional symbolic execution, we require the intermediate state values, between the cycles, to be the same as the initial trace in all cycles except for the cycle in question. This is helpful when a target branch can be reached by the same initial trace, except for the input values in the last cycle. This drastically reduces the search space and cuts down symbolic execution time. However, this will limit our exploration as we will not be able to cover the entire state space. Despite the limitation, we conjecture that the restrictive symbolic execution combined with any random or simulation based test generation framework can still reach most of the reachable branches. This is based on the observation that these branches are often not too distant from the executed paths. Specifically, there may be some branches which require very specific values on registers, which have an extremely low probability to be reached by only simulation based frameworks. Our method can reach those branches very easily.

The central underlying theme in this research is that the information gained during vector generation process for easy-to-reach branches can guide the vector generation for the remaining branches. Our contributions can be summarized as follows:

- For some hard-to-reach or unreached branches an input is generated, for the cycle identified during search process, to reach the branch from an existing execution trace.
- Suggest feasible and infeasible paths to reach a particular branch, useful for search guidance.
- In the process some branches are proved as unreachable.

With experimental results, we show that our technique is able to reach few branches in b14 circuit [22] which have not been reached by any previous methods such as [8-10, 27]. A few

other branches are proved unreachable for b14. For other ITC99 circuits [22], we were able to determine many branches which could be reached in just one cycle from specific cycles of the initial trace.

The rest of this paper is organized as follows: Motivation and related work are presented in Section 4.3. Section 4.4 covers the background tools and terminology. It also discusses the ideas unique to this research, along with the derivation and uses. Section 4.5 gives a detailed methodology of this research, followed by a working example to demonstrate the application to an RTL circuit. This is followed by the description of the algorithm. Section 4.6 presents and discusses the result of this work. Section 4.7 discusses the application of this work. Section 4.8 concludes the paper.

4.3 Motivation and Related Work

In this section we will describe why some branches are hard-to-reach or unreachable and how different techniques have tried to overcome such challenges.

There are many reasons which can make a branch hard-to-reach. One such reason is if the input signal requires a very specific value in a particular cycle. If the input signal has many bits, then the size of the input space can be extremely large. Deterministic methods may be able to generate such values, but it may be computationally intensive to do it if the target branch is deeply nested, as seen in large circuits. In addition, if the specific value required on the input changes for different cycles, then the formulation for deterministic methods would also become more challenging.

A branch may be unreachable if there is a bug in the design which can prevent code execution from reaching certain parts of the code. A branch may also be unreachable if the branch is used to handle any error checking. Such a branch will not be executed in the legal

working of the circuit; hence, it becomes unreachable if the circuit is behaving as expected. Unreachability can also stem from state values which may not be achieved during circuit operation. This can either be due to designer intent or due to lack of optimization in the design. If unreachability of an RTL block can be discovered, it can also point to potential design mistakes.

Simulation based methods such as [9, 23, 24, 26] try to handle hard-to-reach and unreachable cases by letting the vector generation process run for a sufficiently long time. However, they can often fail for hard-to-reach branches and will always fail for unreachable branches, resulting in wasted effort. In such a case, it would be helpful if the test vector generation framework knows if a target branch is reachable during the framework's operation.

Deterministic methods like STAR [10] suffer from path explosion problem as there are too many paths, hence formulae, to be analyzed even for a few cycles of execution. Additionally, symbolic execution by mutating one guard at a time most often leads to branches which have already been explored. HYBRO [8] tries to address the second problem of STAR by mutating only those guards which may lead to new branches. However, it still suffers from the path explosion problem and may be infeasible for large circuits which requires many cycles of unrolling.

Symbolic execution has been applied for branch coverage in [38-40] but based on their methodology, we believe our method would give better results in many cases.

4.4 Preliminaries

In this section, details of code Hierarchy Tree and Leaf Node Path is presented. It is a very important concept in this research work.

4.4.1 Code Hierarchy Tree and Leaf Node Paths

A Verilated C++ code consists of multiple, nested if-else statements, which control the flow to all basic blocks that can be executed in one cycle. A graph, called Code Hierarchy Tree (CHT), can be constructed from it. In this graph, each node represents a basic block of the verilated C++ code. Each node is uniquely identified by a node identifier, which is the counter array index of the branch it represents.

A CHT can be formally defined as follows:

CHT is a structure $< N, E >$ where N is a set of unique nodes, and E is a set of directed edges.

$n \in N : n\ is\ a\ branch\ number$

$E \subseteq N\ x\ N, where\ (n,m) \in E,$ only if the conditional expression to enter block m is one of the immediate statements of basic block identified by *n*.

Figure 4.1 shows a Verilated C++ code and Figure 4.2 shows the corresponding CHT. For the Verilated C++ code listed in Figure 4.1, the variable declarations are removed to save space as they are unimportant in the context of this paper. To generate the CHT, only the relations among the basic blocks are considered. The code is instrumented by adding the array 'cov' to check branch coverage. Each branch is identified uniquely by a branch or coverage number, which is the index of the cov array. The array value is used to check if the execution reached that particular branch, as discussed in Section 2.3. In Figure 4.1, the entry point of the code has a branch number of 0. In the CHT, there is a directed edge from node 0 to nodes 1 and 2 because 'if (s == 0)', the conditional expression to enter blocks 1 and 2, is one of the statements in block 0. Similarly, there is a directed edge from node 0 to nodes identified by 5, 6, and 7; because the conditions to enter blocks 5, 6, and 7 are statements in basic block 0. Similarly, node

2 is connected to nodes 3 and 4. It can be seen that node 0 is not connected to node 3 because the condition to enter block 3 is not immediately from block 0, but from block 2.

```
V_C_func()
{ ++cov[0];
  datai = input();
  s=((ir/0x2000)%3);
  mf=((ir/0x800)%2);
  ir = ir * (-1);
  if(s == 0)
  {  ++cov[1];
     r = reg0;
  }
  else
  { ++cov[2];
    if(s == 1)
    { ++cov[3];
      r = reg1;
    }
    else
    { ++cov[4];
      r = reg2;
    }
  }
  if(mf == 0)
  { ++cov[5];
    m = tail;
  }
  else
  { ++cov[6];
    m = datai;
  }
  if(ir > 0)
  { ++cov[7];
    if(r == m)
    { ++cov[8];
    }
    else
    { ++cov[9];
    }
  }
}
```

Figure 4.1: Example Verilated C++ program

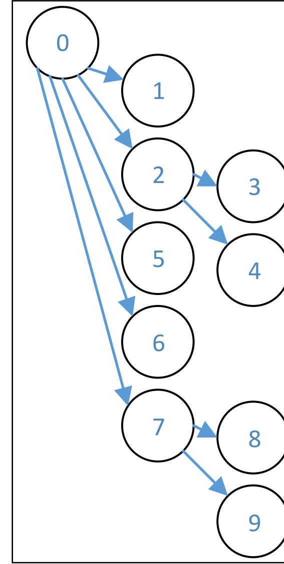

Figure 4.2: Code Hierarchy Tree for verilated code of Figure 4.1

It can be seen that each path in the CHT starts from the same node which is the entry point of the verilated C++ code. It is hereafter referred to as the root node of the CHT. Each path in CHT ends at a node which represents a branch that does not contain any more conditional expressions inside it. It is referred as a leaf node of the CHT. Each path from the root to a leaf

node in the CHT is called a leaf node path. This path is similar to an execution path of the circuit for a single cycle, but an execution path can have multiple leaf node paths. Multiple leaf node paths are present in the execution path if there are parallel execution blocks in the Verilog code. Such blocks are easy to identify in Verilog code as these blocks begin with 'always' statement. Thus, in any cycle of circuit simulation, at least one of the leaf node paths will be completely traversed.

To extract leaf node paths, we start from the root node and traverse until we reach a leaf node. By this method all the possible leaf node paths for Figure 4.1 are

(0, 1); (0, 2, 3); (0, 2, 4); (0, 5); (0, 6); (0, 7, 8); and (0, 7, 9)

This can also be inferred from CHT shown in Figure 4.2.

A leaf node path also expresses the dominator relation among the basic blocks of the code. i.e., to reach a particular basic block in a given cycle, which other basic blocks should definitely be traversed.

If the target branch is not a leaf node, then there may be multiple leaf node paths that contain the target branch or node. However, all the nodes prior to the target node in each of those leaf node paths will always be the same. It is because there is only one entry point to each of the nodes in the CHT as a block can be entered from only one another block if the code consists of multiple, nested if-else statements. In such a scenario, the parent of each node would be unique and a recursive backward traversal will always lead to the same set of predecessor nodes. For example, in Figure 4.2, node 7 is on two leaf node paths: (0, 7, 8), and (0, 7, 9). Even though node 7 is on two paths the predecessor of node 7 is always the same, which is node 0 in this case.

4.4.2 Execution path

An execution path is a sequence of branches that are executed in a particular cycle. It is dependent on both the register values at the beginning of a cycle and the circuit input values applied for the particular cycle. An execution path may contain one or more leaf node paths due to the parallel nature of hardware. In Figure 4.1, it can be seen that after the first if-else construct (branches 1 and 2) is executed in block 0, the execution will move to the next set of if-else construct (branch 5 and 6), and then to the next if statement (branch 7). This way, the execution path will have multiple leaf node paths. For the code listed in Figure 4.1, a non-exhaustive list of execution paths is as follows:

0, 1, 5; 0, 1, 6; 0, 2, 3, 5, 7, 9; 0, 1, 5, 7, 8;

4.5 Methodology

The first step in the proposed method is to obtain an execution trace by simulating an input sequence. This input sequence is also referred as initial sequence henceforth. The generation of this initial sequence can be random or from a designer, which is not part of this work. From the simulation, an execution trace is obtained. The execution trace contains the register values, executed branch numbers and the inputs applied for each cycle. This execution trace is used as an input to our program. The branches that are reached can be checked in the execution trace. All the leaf node paths are extracted from the verilated C++ code, as described in Section 4.4.1. We pick one of the branches yet to be reached as the target branch. In selecting the target branch, it is preferred if the target branch is an immediate child of a reached branch in the CHT. The idea behind selecting an immediate unreached branch as the target branch is that

we explore the unreached branches one level at a time. This strategy can recursively resolve the reachability of all branches gradually, from the reached branches to the unresolved ones.

Next, we obtain the cycles in which the restrictive symbolic execution is to be performed. In order to do so, the leaf node paths are observed to find a path where the target branch is present. We look for the cycles in which the immediate preceding branch of the target branch in the path is reached. The execution trace is checked to find if a branch is reached or not. If the immediate preceding branch is not reached, then the leaf node path is traversed backwards until we find one preceding branch which is reached. The cycles in which that preceding branch is reached are identified. The target branch has a high probability of being reached in this set of identified cycles. From this set, one cycle is selected at a time as a target cycle. For each target cycle, the restrictive symbolic execution is performed. If, in any target cycle, a satisfiable solution is found, then the analysis on the remaining cycles is aborted. Also, the operation is aborted for a particular target branch if no satisfiable solution is found after all the cycles in the set of identified cycles have been analyzed. Once a target cycle is selected, the program is flattened.

Flattening the program means extracting the executed statements in the verilated C++ code in the order it was executed for a particular cycle. Flattening requires a target cycle number, branches executed in that cycle and an unreached target branch. The execution trace contains the indices of all branches executed in a particular cycle, which helps to identify the statements executed in that cycle. Flattening is performed from the beginning of execution for the particular cycle until the activating condition of the target branch is reached. The first statement executed in the target cycle is the first statement of the flattened program, while the activating condition of the target branch in the same cycle is the last statement of the flattened program. Any statement

executed after the activating condition is not considered. It is to be noted that the activating condition was not evaluated to 'true' during the simulation, because the target branch was not reached. Therefore, the flattened program is a set of statements and a few conditions which closely represents the execution path which will lead to the target branch in the target cycle. After flattening is completed, the frontier statements and frontier signals in the program are extracted, which is described next.

Frontier statements are a subset of flattened program statements which influence the values of the signals in the activating condition of the target branch. To keep track of all the signals which influence the signals in the activating condition, a list called "frontier signals" is constructed. The signals involved in the activating condition are set as the initial members of the frontier signals. The activating condition itself is the first member of the frontier statements. To extract the frontier statements, the flattened program is traversed backwards from the activating condition to the top of the flattened program. During the traversal, if a statement is encountered where any of the current frontier signals is assigned a value, then that statement becomes part of frontier statements and the frontier signals are updated by removing the assignee signal and adding the assignor signal. If the assignor is a constant value, then no assignor signal is present. Hence, no signal is added to the frontier signals. The extraction of frontier statements continues until the top of the flattened program is reached.

Next, we use an SMT solver to find a value for the circuit's input signal in the target cycle. In this stage, the activating condition, selected preceding conditions and frontier statements are added to the SMT solver instance. Among the preceding conditions, only those involving frontier signals are added to the SMT instance. If the instance is satisfiable, then the statements of the frontier statements are incrementally added as constraints to the SMT instance.

If, after adding any statement, the instance becomes unsatisfiable, then the analysis is aborted for the particular target cycle and the path is treated as infeasible. However, upon adding all the frontier statements to the SMT solver, if the instance is still satisfiable and one of the frontier signals is an input signal of the circuit, then additional constraints are added. The additional constraints are the concrete values of the frontier signals which are not input signals of the circuit. These signals' values are obtained from the previous cycle in the execution trace. In other words, for each non-input frontier signal, the corresponding value of the signal is added as an additional constraint to the solver instance. Next, the instance is solved. If the instance is still satisfiable then a model is generated to get the value required on the circuit's input signal. The value returned by the solver when applied to the circuit's input helps the execution reach the target branch from the starting state of the analyzed cycle.

As this analysis is conducted only for the frontier statements of one cycle, the analysis involves very few statements compared to all the statements executed in that cycle. Also, all the statements will be included just once, because verilated C++ code doesn't have loops in a single cycle as discussed in Section 2.3.

In order to reach a target branch, the only change we need to make to the initial sequence is that we suggest a different input value in the target cycle than the one which was originally present. All the inputs after the suggested cycle are discarded in the vector. This change in input has a good probability in leading the execution to the target branch. However, it is possible that changing the input signal value may alter the execution path itself and the branch may not be reached. This case is handled by trying the same analysis for a different target cycle. It is still possible that no conclusions about reachability can be made after all the possible target cycles have been analyzed. In all the designs that we simulated, we did not find a case where we could

not reach the target branch if it met the conditions to be reachable in our analysis, i.e., the SMT instance is satisfiable and one of the frontier signals is an input.

In some cases, we can determine if a branch is unreachable, if there are assignment(s) just before the target condition, which definitely need to be executed irrespective of the path taken to reach the target condition. If the assignment(s) is conflicting with the target condition then the branch is unreachable. Two branches in b14 have this property and hence they are proved to be unreachable.

4.5.1 A Working Example

The code listed in Figure 4.1 is a very simplified version of a CPU design where resolving reachability of one of the branches is very difficult as it requires a very specific input value in one cycle. A snippet of the execution trace obtained for the circuit is

```
Cycle = 100, which covers: (0, 2, 3, 6, 7, 9), ir = 30, reg1 = 24, datai = 7, m = 7, r = 24, s = 1, mf = 1
```

It can be seen that reaching branch 8 is difficult because it requires that two 32-bit registers, namely ‘`r`’ and ‘`m`’, to be exactly equal in a particular cycle.

For the target branch 8, the leaf node path is (0, 7, 8). The execution trace is examined to find the cycles in which the parent of target branch, i.e., branch 7, has been reached. From the snippet of execution trace, it can be seen that branch 7 is reached in cycle 100. For cycle 100, the flattened program is constructed by inspecting the branches reached in that cycle, until the activating condition of the target branch. The instrumentation for branch coverage is discarded while extracting the flattened program because the instrumentation does not affect the working of the circuit. We start the analysis from the beginning of the cycle. The first statement of the cycle (‘`datai = input();`’) becomes the first statement of the flattened program and the

analysis continues till the activating condition (‘`(r == m)`’) is reached. The branches reached in cycle 100 are (0, 2, 3, 6, 7, 9). Therefore, flattened program in this case is as follows:

Coverage Index	Statements
0	`datai = input();`
	`s = ((ir/0x2000) % 3);`
	`mf = ((ir/0x800) % 2);`
	`ir = ir * (-1);`
2	None
3	`r = reg1;`
6	`m = datai;`
7	None
and the target condition	‘`(r == m)`’

From the flattened program, the frontier statements and frontier signals are extracted in reverse order to best reflect the influence on the target condition. Frontier statements include all the statements in the flattened program which affect the target condition. The frontier statements are added incrementally to the SMT solver instance. The frontier signals, frontier statements, and the corresponding SMT clause as updated with each statement are as follows:

Frontier Signals	Statements	SMT clauses
r, m	r == m	r == m
r, datai	m = datai	m == datai
reg1, datai	r = reg1	r == reg1
reg1, input()	datai = input()	datai == input()

Here, we see that in the final frontier signals, there is one input signal and the other signal is a register. In order to generate a value for the input in cycle 100, we look for the non-input signal value in the previous cycle, i.e., the value of register reg1 in cycle 99. This value is added to the SMT instance as an additional constraint, i.e., (reg1 == value_of_reg1_in_cycle_99) The value for input() is obtained by solving this instance. With this example in mind, the next section discusses the generic algorithm of this method.

4.5.2 Algorithm

The algorithm of our approach i.e., cycle-by-cycle restrictive symbolic execution is shown in Algorithm 4

Algorithm 4: Algorithm of cycle-by-cycle Restrictive Symbolic Execution

```
1  Initialize, Extract all leaf node paths
2  for each target branch do
3      Get target cycles, the cycles in execution trace where parent
       branch was reached.
4      for each of the target cycles do
5          Extract flattened program, activating and preceding conditions
6          Extract frontier statement and frontier signals
7          Initialize the SMT instance, add the activating condition and
           relevant preceding conditions
8          for each statement in frontier statement do
9              Add the statement to solver
10         if (SMT instance == SAT)&&(frontier ∩ input sig ≠ φ)
           then
11             Add non-input signal value from previous cycle as
               constraint
12             solve and generate value of input
```

4.6 Results

The experiments were run on a laptop with 8 GB RAM, Intel® Core™ i5-3210M CPU@2.5 GHz running Windows 7, 64 bit operating system. Code was implemented using Python version 2.7.5 and was run on one of the four processors. We perform our analysis on ITC99 [10] benchmark circuits.

We generated random input sequence for each RTL circuit. The execution trace from this initial sequence was used as the initial execution trace on which our technique was applied. For each target branch, we apply our analysis on a maximum of 20 target cycles. It is because we found that after a few target cycles; there are no new paths to uncover and hence doing an analysis for more target cycles wastes computational resources. Our analysis returns the target coverage number, target cycle number and the values for the inputs for the target cycle. For each of the target coverage, we compose a vector. The inputs in the vector until the cycle preceding the target cycle are the same as the initial vector. The input for the target cycle is the value generated by our analysis. During our experiments, we append a few more random inputs for a few more cycles. It is because the newly reached branch can contain assignments to reach few more branches. Each such vector, created above, is appended to the initial vector after asserting the 'reset' input of the circuit, so that it starts from a fixed state.

We used both short and long initial random sequence to test the usefulness of our approach. The results of the analysis are shown in Table 4.1. The first column lists the name of the circuit. The second column lists the total number of branches in the circuit. Columns 3, 4 and 5 report the results for the short initial sequence; column 3 lists the number of branches reached with the short initial sequence with the length of the sequence in parentheses, column 4 lists the number of new branches reached by our method using the short initial sequence, and column 5

lists the time taken for the analysis. Columns 6, 7 and 8 list the details when our analysis was applied to long initial sequence with length 50,000.

Table 4.1: Result of application of our method on short and long random vector sequence

Cir cuit	Total branches	Short initial sequence			Long initial sequence, length = 50,000		
		Coverage (sequence length)	Extra branches covered	Time (s)	Coverage	Extra branches covered	Time (s)
b01	26	13 (8)	11	22.6	25	1	22.3
b06	24	12 (9)	5	10.3	22	1	25.9
b07	20	11 (7)	2	5.1	17	1	39.0
b10	32	16 (10)	6	12.3	26	5	64.1
b11	33	16 (10)	1	18.4	30	1	32.1
b12	105	54 (217)	14	138	61	12	272
b14	211	105 (122)	29	149	193	4	145

This analysis shows that when we start with a short initial sequence, our method can help to identify input cycles where we can modify inputs to reach better coverage. For example, in circuit b12 which contains 105 branches, the short initial sequence of 217 vectors can reach 54 branches. With our approach, we can reach another 14 branches. On the other hand, when 50K random vectors were applied, 61 branches are reached by the random vectors. Note that 61 is less than what we had achieved (54+14) with the short sequence. Nevertheless, starting with this long initial sequence, we can still reach 12 more branches. The results for other circuits can be explained in a similar manner.

Circuit b14 has three branches which have a similar kind of dependency as the code presented in Figure 4.1, i.e., hard-to-reach branches requiring unique values in specific cycles and the values change for each cycle. These three branches have not been reached by other methods reported to date.

Among the best coverages reported by state-of-the-art methods [8-10, 27], at least 17 branches were unresolved for b14. We applied our analysis on all the unresolved branches by BEACON [9]. The input sequence shared by BEACON was used as the initial sequence. The vectors generated by our restricted symbolic execution were composed as one vector, with the method discussed in the beginning of this section, i.e., each suggestion was composed as one input sequence and all such input sequences were appended one after the other in the initial sequence. This newly generated input sequence was simulated and its result is shown in Table 4.2. Out of the 17 unreached branches, our method suggested inputs to reach 3 branches and proved 2 as unreachable. The simulation with the final input sequence was able to reach all 3 of those suggested branches.

Table 4.2: Result of application of our method for branches unresolved by BEACON for b14

Circuit	Total Branches	Branches unresolved by BEACON [9]	Out of unresolved, generated inputs to reach, by our method	Out of unresolved, proved unreachable, by our method
b14	211	17	3	2

4.7 Application of our work

Given any vector sequence, we can suggest changes to inputs in certain cycles in order to obtain (1) an increase in coverage, (2) a reduction in test set size, and (3) resolve unreachability.

1) Increase coverage – If a certain hard-to-reach branch has not been reached with the original input sequence, then with this method we can add the needed input at a specific cycle and improve the coverage.

2) Reduce test set size – In simulation based methods, it is possible that there may be unnecessary looping around reached states before reaching a new state which unlocks entry into

new branch. In such cases, it would be helpful if there is a suggestion for an input in the cycle, which can help reach the branch in an earlier cycle, so that the unnecessary traversal of the reached states can be avoided.

The proposed method can help to reduce the test set sizes by offering the suggested vectors that can reach the desired branches. Such help can have a secondary effect of reducing the post-test-set-compaction efforts such as [41, 42]

3) Resolve unreachability - In simulation based methods, if branches have been proved unreachable, then they can be trimmed from search space leading to saved time and effort in trying to generate vectors for them.

4.8 Conclusions

We proposed a restricted symbolic execution which is able to reach hard branches even with an initial short random sequence. More importantly, it can reach some branches which have not been reached by other methods. It can also prove some branches as unreachable during the search process. Adding this method to simulation-based frameworks can resolve the reachability of most branches in the circuit. This work can also help improve the coverage of a given sequence. The computational cost of this method is smaller than conventional symbolic execution, making it feasible for cases where pure symbolic execution may not have been feasible. It can also improve the overall test vector quality with a reduced test length.

5 Conclusion and Future Research Ideas

Test stimuli generation for RTL circuit validation has still not reached a point where efficient stimuli can be generated for all circuits. We looked on few of the reasons for this deficiency and tried to address them. One of the reasons is that some branches in RTL code are unreachable and test stimuli generators may put lot of effort in generating test stimuli for such cases, leading to wasted effort. In this regard, we proposed a method to quickly determine reachability of each branch. If a branch is determined unreachable, it can be trimmed from the search space of the test stimuli generation framework in the beginning itself. Additionally, it can also determine the necessary branches which must be executed to reach a target branch. This can serve as guidance for the framework to explore a fewer set of paths to reach a target branch.

Few branches are hard-to-reach because they require a specific input value in a specific cycle. Such branches are almost never reached by stimuli generated by simulation based stimuli generation frameworks, because it generates input without any knowledge of the characteristics of the circuit. We proposed a method of cycle-by-cycle restrictive symbolic execution to help such input value generation. Our method builds up on the information gained during the simulation phase to identify cycles in which a target branch has good probability of being

reached. Once the cycle is identified, a symbolic analysis is done on the code executed in that cycle in a restrictive manner to reduce the computational cost. The proposed method was able to reach a few more branches which have not been reached by other published methods on ITC99 circuits.

Both of these research methods have increased the branch coverage for ITC99 circuits. For some circuits, where all branches were not covered, combining both the methods resolves the reachability of each branch. Thus, resolving reachability of each branch in the circuit, taking the branch resolvability to 100% for those circuits.

While doing this research work, there were a few more ideas which might be interesting to explore and may directly or indirectly improve this research work. They are:

1. Both the research work can be integrated in a simulation based test stimuli generation framework to add intelligence to the framework. This would help the framework to avoid wasting effort for stimuli generation for unreachable branches and also generate input for few hard-to-reach branches quickly.
2. Branch coverage has become a standard measure to describe the effectiveness of a test stimulus. However, simple branch coverage doesn't differentiate between hard-to-reach and easy to reach branch, giving both of them the same weightage. Thus, a stimulus which reaches more hard-to-reach branches but may not reach all the easy to reach branch is interpreted as a poor stimulus, when compared to another stimulus which reaches much more easy-to-reach branches but very few hard-to-reach branches.

   Hence, some alternate measures may be researched which considers the difficulty to reach a branch along with the numbers of branches reached. This measure may also be helpful in guiding the test stimuli generation frameworks to pick a target branch to generate stimuli for.

3. In the restrictive symbolic execution work, only one time frame is considered for symbolic execution, it may be interesting to explore if this analysis can be explored for few more previous time frames.
4. In the restrictive symbolic execution, the paths that have been explored can be saved. So that, in the next cycles to be analyzed, if the same path is found then the analysis for that path can be discarded and no analysis needs to be done on it, saving computational cost.
5. In this research work, for doing the source-to-source transformation, regex is heavily used. It can be explored to do the transformation using a compiler engine. This will allow more valid C++ constructs to be transformed into clauses that can be added in z3 instance.

6 Bibliography

[1] *The Chip that Jack Built*. Available: http://www.ti.com/corp/docs/kilbyctr/jackbuilt.shtml

[2] P. Patra, "On the cusp of a validation wall," *Design & Test of Computers, IEEE,* vol. 24, pp. 193-196, 2007.

[3] H. Foster, "The 2010 Wilson Research Group Functional Verification Study," ed, 2011.

[4] K. Gent and M. S. Hsiao, "Dual-Purpose Mixed-Level Test Generation Using Swarm Intelligence," in *Test Symposium (ATS), 2014 IEEE 23rd Asian*, 2014, pp. 230-235.

[5] L. Kuan-Yu, C. Po-Juei, L. Ang-Feng, J. C. M. Li, M. S. Hsiao, and W. Laung-Terng, "GPU-based timing-aware test generation for small delay defects," in *Test Symposium (ETS), 2014 19th IEEE European*, 2014, pp. 1-2.

[6] S. Bagri, K. Gent, and M. S. Hsiao, "Signal Domain Based Reachability Analysis in RTL Circuits," in *16th International Symposium on Quality Electronic Design (ISQED)*, Santa Clara, CA, USA, 2015.

[7] S. Prabhu, V. V. Acharya, S. Bagri, and M. S. Hsiao, "Property-checking based LBIST for improved diagnosability," in *Test Symposium (ETS), 2014 19th IEEE European*, 2014, pp. 1-2.

[8] L. Lingyi and S. Vasudevan, "Efficient validation input generation in RTL by hybridized source code analysis," in *Design, Automation & Test in Europe Conference & Exhibition*, 2011, pp. 1-6.

[9] M. Li, K. Gent, and M. S. Hsiao, "Design validation of RTL circuits using evolutionary swarm intelligence," in *IEEE International Test Conference (ITC)*, 2012, pp. 1-8.

[10] L. Lingyi and S. Vasudevan, "STAR: Generating input vectors for design validation by static analysis of RTL," in *IEEE International High Level Design Validation and Test Workshop*, 2009, pp. 32-37.

[11] *Verilator Home Page*. Available: http://www.veripool.org/wiki/verilator

[12] S. Prabhu, M. S. Hsiao, L. Lingappan, and V. Gangaram, "Test generation for circuits with embedded memories using SMT," in *Test Symposium (ETS), 2013 18th IEEE European*, 2013, pp. 1-1.

[13] S. Prabhu, M. S. Hsiao, L. Lingappan, and V. Gangaram, "A SMT-based diagnostic test generation method for combinational circuits," in *VLSI Test Symposium (VTS), 2012 IEEE 30th*, 2012, pp. 215-220.

[14] L. Moura, N. Bj\, \#248, and rner, "Satisfiability Modulo Theories: An Appetizer," in *Formal Methods: Foundations and Applications*, V. Marcel, O. cius, and W. Jim, Eds., ed: Springer-Verlag, 2009, pp. 23-36.

[15] C. Barrett, M. Deters, L. Moura, A. Oliveras, and A. Stump, "6 Years of SMT-COMP," *J. Autom. Reason.,* vol. 50, pp. 243-277, 2013.

[16] *Z3 Home Page*. Available: http://z3.codeplex.com/

[17] *Wikipedia Page on Abstract Syntax Tree*. Available: http://en.wikipedia.org/wiki/Abstract_syntax_tree

[18] E. Bendersky. *Parser for C in python*. Available: https://github.com/eliben/pycparser

[19] *BSD 2-clause license*. Available: http://opensource.org/licenses/bsd-license.php

[20] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," presented at the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, San Diego, California, USA, 1988.

[21] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.,* vol. 13, pp. 451-490, 1991.

[22] S. Davidson, "ITC'99 Benchmark Circuits - Preliminary Results," in *International Test Conference*, 1999, pp. 1125-1125.

[23] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Sequential circuit test generation using dynamic state traversal," in *European Design and Test Conference*, 1997, pp. 22-28.

[24] 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 *VLSI Design, 1997. Proceedings., Tenth International Conference on*, 1997, pp. 475-481.

[25] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Dynamic state traversal for sequential circuit test generation," *ACM Trans. Des. Autom. Electron. Syst.,* vol. 5, pp. 548-565, 2000.

[26] W. Weixin and M. S. Hsiao, "Efficient Design Validation Based on Cultural Algorithms," in *Design, Automation and Test in Europe*, 2008, pp. 402-407.

[27] K. Gent and M. S. Hsiao, "Functional Test Generation at the RTL Using Swarm Intelligence and Bounded Model Checking," in *22nd Asian Test Symposium*, 2013, pp. 233-238.

[28] Q. Xiaoke and P. Mishra, "Scalable Test Generation by Interleaving Concrete and Symbolic Execution," in *13th International Conference on VLSI Design and 27th International Conference on Embedded Systems*, 2014, pp. 104-109.

[29] M. Sauer, S. Kupferschmid, A. Czutro, S. Reddy, and B. Becker, "Analysis of Reachable Sensitisable Paths in Sequential Circuits with SAT and Craig Interpolation," presented at the 25th International Conference on VLSI Design 2012.

[30] *A Free, Fast and Small Automatic Formatter for C, C++, C++/CLI, C#, and Java Source Code*. Available: http://astyle.sourceforge.net/

[31] A. Biere, E. Clarke, R. Raimi, and Y. Zhu, "Verifying Safety Properties of a PowerPC− Microprocessor Using Symbolic Model Checking without BDDs," in *Computer Aided Verification*. vol. 1633, N. Halbwachs and D. Peled, Eds., ed: Springer Berlin Heidelberg, 1999, pp. 60-71.

[32] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," *Form. Methods Syst. Des.,* vol. 19, pp. 7-34, 2001.

[33] Z. Liang, M. R. Prasad, and M. S. Hsiao, "Incremental deductive & inductive reasoning for SAT-based bounded model checking," in *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, 2004, pp. 502-509.

[34] M. Elbayoumi, M. S. Hsiao, and M. ElNainay, "Selecting critical implications with set-covering formulation for SAT-based Bounded Model Checking," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, 2013, pp. 390-395.

[35] M. Elbayoumi, M. S. Hsiao, and M. ElNainay, "Set-cover-based critical implications selection to improvesat-based bounded model checking: extended abstract," presented at

the Proceedings of the 23rd ACM international conference on Great lakes symposium on VLSI, Paris, France, 2013.

[36] A. F. Donaldson, L. Haller, D. Kroening, and P. R, "Software verification using k-induction," presented at the Proceedings of the 18th international conference on Static analysis, Venice, Italy, 2011.

[37] J. C. King, "Symbolic execution and program testing," *Commun. ACM,* vol. 19, pp. 385-394, 1976.

[38] S. Prabhu, M. S. Hsiao, S. Krishnamoorthy, L. Lingappan, V. Gangaram, and J. Grundy, "An Efficient 2-Phase Strategy to Achieve High Branch Coverage," in *Test Symposium (ATS), 2011 20th Asian*, 2011, pp. 167-174.

[39] S. Krishnamoorthy, M. Hsiao, and L. Lingappan, "Strategies for scalable symbolic execution-driven test generation for programs," *Science China Information Sciences,* vol. 54, pp. 1797-1812, 2011/09/01 2011.

[40] S. Krishnamoorthy, M. S. Hsiao, and L. Lingappan, "Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs," in *Test Symposium (ATS), 2010 19th IEEE Asian*, 2010, pp. 59-64.

[41] M. S. Hsiao and S. T. Chakradhar, "State relaxation based subsequence removal for fast static compaction in sequential circuits," in *Design, Automation and Test in Europe, 1998., Proceedings*, 1998, pp. 577-582.

[42] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Fast Static Compaction Algorithms for Sequential Circuit Test Vectors," *IEEE Trans. Comput.,* vol. 48, pp. 311-322, 1999.