

Increasing Branch Coverage with Dual Metric RTL Test Generation

Kunal Bansal

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 Hsiao, Chair
Haibo Zeng
A. Lynn Abbott

June 14, 2018
Blacksburg, Virginia

Keywords: Mutation, Coverage, Metric, RTL

Copyright 2018, Kunal Bansal

Increasing Branch Coverage with Dual Metric RTL Test Generation

Kunal Bansal

(ABSTRACT)

In this thesis, we present a new register-transfer level (RTL) test generation method that makes use of two coverage metrics, Branch Coverage, and Mutation Coverage across two stages, to cover hard-to-reach points previously unreachable. We start with a preprocessing stage by converting the RTL source to a C++ equivalent using a modified Verilator, which also automatically creates mutants and the corresponding mutated C++ design, based on arithmetic, logical and relational operators during conversion. With the help of extracted Data Dependency and Control Flow Graphs, in every <golden, mutation> pair, branches containing variables dependent on the mutated statement are instrumented to track them. The first stage uses Evolutionary algorithms with Ant Colony Optimization to generate test vectors with mutation coverage as the metric. Two new filtering techniques are also proposed which optimize the first stage by eliminating the need for generating tests for redundant mutants. The next stage is the original BEACON which now takes the generated mutation test vectors as the initial population unlike random vectors, and output final test vectors. These test vectors succeed in improving the coverage up to 70%, compared to the previous approaches for most of the ITC99 benchmarks. With the application of filtering techniques, we also observed a speedup by 85% in the test generation runtime and also up to 78% reduction in test vector size when compared with those generated by the previous techniques.

Increasing Branch Coverage with Dual Metric RTL Test Generation

Kunal Bansal

(GENERAL AUDIENCE ABSTRACT)

In the recent years, Verification has become one of the major bottlenecks in integrated circuit design process, which is exacerbated by the increasing design complexities today. Today designers start the design process by abstracting the initial design in a manner similar to software programming language using a higher abstraction language called Hardware Descriptive Language(HDL). Hence, an HDL based design also contains a number of case statements and if-else statements, also called *branches*, similar to a software design. *Branches* indicate decision points in the design and high branch coverage based tests can give us an assurance that the design is properly exercised as compared to those given by randomly generated tests. In this thesis, we introduce a new test generation methodology which generates tests using the help of user introduced mutants to ensure higher branch coverage. Mutation testing is similar to a fault testing method, in which an error or a fault is deliberately introduced into the design and we check if the tests generated are able to detect the fault. An important property of a mutant is that: when a mutant is applied and if the mutated part of the design is exercised by the given test suite, then the following data and control flow path taken can be different from that taken on the original design. This important property along with proper guidance is used in our work to reach some branches which are difficult to cover by random test vectors, and this is the main basis of this thesis. Applying this method, we observed that the branch coverage increased with a decrease in test generation runtime and test vector length when compared to previously proposed techniques.

Dedication

Dedicated to my family.

Brother Sourabh Bansal

Parents Anil Kumar and Neetu Bansal

Acknowledgments

Foremost, I owe my greatest thanks to my advisor Dr. Michael Hsiao for his continuous guidance, patience and motivation throughout the duration of my research. I am very fortunate to get an opportunity to interact with my advisor through his course "Electronic Design Automation" in 2016, where his teaching style, the class projects and the overall experience inspired me to pursue my Master thesis under his guidance. His office door was always open and very prompt in helping me, whenever I had even a silly question regarding my research or thesis writing. I would like to thank him for always encouraging me to build a mindset of finding solutions to the problems by myself through extensive research and maintaining high expectations throughout my thesis.

I would also like to thank Dr. Haibo Zheng and Dr. Amos L Abbott for their kindness in agreeing to serve on my thesis committee. I am also grateful to my ECE Advisor Mrs. Mary Brewer for her support in helping clear any doubt I had related to paperwork throughout my duration of my stay at Virginia Tech. I also thank to the staff of Graduate School for their timely help regarding any administrative issues or questions I faced.

Working in lab would have never been fun, if I haven't had the company of my PROACTIVE lab members : Tania Khanna, Akash Agrawal, Sonal Pinto, Aravind Krishnan, Yue Zhan, and Tonmoy Roy. I am grateful to Akash for his constant encouragement and patience in listening to my silliest questions on my research work. I thank Tania and Aravind for being my amazing coffee buddies and being patient enough to handle my bad jokes and helping me out by sharing their viewpoints whenever I got stuck during research. I specially thank

Tania for supporting me over the last few weeks of my thesis, without who it wouldn't have gone smoothly.

The 2.5 years stint at Virginia Tech would have not been possible without my friends I made here: Aakanksha, Tania, Shruti, Abhishek, Surabhi, Akshay, Shamit, Nishant, Anup, Suresh, Shailesh, Shivam, Aditya, Prakriti, and Vinit. The dance nights, music nights, group dinners and many more with these folks just made my graduation life as one of my best moments, which I will always cherish. I thank Abhishek for making sure that I do not skip to miss meals and being always there to help me whenever I needed it. I am especially grateful to Aakanksha, for always being around me even during my difficult times at Virginia Tech and constantly encouraging me to believe that "I can do it".

Last but not the least, I am very thankful for the immense love and affection from my family who were my ultimate pillar of support. Their motivation and encouragement were the first and foremost reasons today, I got a chance to come to Virginia Tech and experience this graduate school life.

Kunal Bansal

Blacksburg

June 14, 2018

Contents

List of Figures	x
List of Tables	xii
1 Introduction	1
1.1 Problem Scope	1
1.2 Contributions of the Thesis	3
1.2.1 Dual Metric Test Generation Framework	3
1.2.2 Mutant Filtering	6
1.3 Thesis Organization	7
2 Background	8
2.1 Genetic Algorithm	8
2.2 Ant Colony Optimization	10
2.2.1 Mutants	11
2.3 RTL Instrumentation	12
2.4 Data Dependency Graph	14
2.5 Control Flow Graph	16
2.6 Related Work	16

3	Dual Metric RTL Test Generation	19
3.1	Framework & Methodology	19
3.1.1	Preprocessor	21
3.1.2	First Pass Test Generation(TG)	26
3.1.3	Second Pass Test Generation	29
3.2	Experiments & Results	30
3.2.1	Tool Settings	31
3.2.2	Branch Coverage	32
4	Mutant Filtering	34
4.1	First Mutation Filtering Approach	36
4.2	Second Mutation-Filtering Approach	38
4.3	Experiments & Results	39
5	PCB Simulation	44
5.1	Motivation	44
5.2	Methodology	47
5.3	Results	49
6	Conclusion	50
6.1	Limitations and Future Work	50
6.2	Conclusion	51

List of Figures

2.1	Phases in Genetic Algorithms	9
2.2	Verilog HDL Snippet and its corresponding C++ verilated model	12
2.3	Example Patch file for a sample mutant	13
2.4	Example of verilated RTL with its DDG	14
2.5	CFG for the above verilated RTL	15
3.1	Example Verilog to Justify Methodology	20
3.2	Preprocessing for Mutation Coverage Guidance Search	22
3.3	Example Patch file for a sample mutant	23
3.4	Example Verilog HDL Design and Instrumented C++ Model with Mutation Coverage and Branch Coverage Counters	24
3.5	First Stage of Dual Metric Test Generation	26
3.6	Second Stage of Dual Metric Test Generation	30
4.1	Example of verilated RTL with its CFG and DDG	35
4.2	Snippet of Verilated RTL and its CFG	37
5.1	Simple PCB	45
5.2	Methodology	47

5.3 Post Processing Flow	48
------------------------------------	----

List of Tables

3.1	ITC99 Benchmarks	30
3.2	Branch Coverage	31
3.3	Mutation Coverage	33
4.1	ROR Mutants and Input to Differentiate for Original: $z < 0$	40
4.2	Benchmark Characteristics	40
4.3	Comparison of Different Approaches of Mutation Filtering	41
5.1	Branch Coverage for PCB Simulations	49

Chapter 1

Introduction

The New York Times [1] once stated: "The integrated circuit, better known as the semiconductor chip, has unleashed change comparable to the Industrial Revolution by making the computer revolution and the digital age possible." Today there are far more chips on earth than people. We have so many modern-day gadgets which has been very useful to human livelihood like phones, automobiles, laptops, electronic equipments, etc. and all of the credit goes to this computer revolution and the growth in the digital era. Critical products and services like spacecrafts, health services, security, etc. are built with these chips and even a small flaw in the design or its functionality can cause a significant loss to both people's lives and money. Today, the reputation of companies using chips vastly depends on the reliability of their designs and a single mistake can also bankrupt the company. Hence, verifying that the products made are error-free, is the utmost priority in any product building cycle, before it is brought to the market.

1.1 Problem Scope

Decades ago, designs were not that complex. They had small numbers of logic gates and interconnections which could be verified manually by engineers [12]. But today, as per Moore's Law [27], there has been an exponential increase in design complexities. These complex designs can be thought of as a group of interconnected modules and IPs (Intellectual Properties)

wherein multiple designers start designing by laying out the specification at a higher level of abstraction (Register Transfer Level) for individual modules/IPs. Such designs consist of millions of gates and demand tremendous verification resources in the chip production cycle. Verification is one of the major bottlenecks in today's integrated circuit design process. Verification of these designs at an early level of design abstraction can help reduce the likelihood of bugs arising in later stages. One of the most common verification methodology used at the RTL level is Simulation based verification, which involves generating and applying stimuli to the design. A popular metric to analyze the quality of the verification is Code Coverage [9] which measures the number of executed statements/branches/functional-features/Assertions/etc. These coverage metrics are adopted from the software verification metrics, i.e., branch coverage, statement coverage, assertion coverage, etc. [4]. With the growing increase in complexity of designs, RTL designers could face the following issues :

1. The number of branches and their complexity could increase, and
2. The probability of human-induced faults while writing the RTL increases.

An effective metric to analyze and track the coverage of the first issue above is Branch coverage. It measures the percentage of branches of the source code (RTL) covered while simulating a test suite. One of the easiest approach towards high branch coverage closure, would be generating vectors and track the coverage. Random stimuli, though popular, do not make use of all the details of the functional design information at this higher abstraction level. Consequently, they do not achieve high coverages. To bridge the gap, researchers proposed techniques to reach high branch coverage by generating test vectors. But these vectors would either take many man-hours to generate or they may not access all the branches since some areas of the design might require some specific sequence of vectors, which are difficult to generate.

Recently, BEACON [24] exploited Evolutionary Algorithms for test generation to reach a high branch coverage. The initial population is filled with random vectors and fitness focuses on reaching the uncovered branches while simulating the circuit over a number of cycles. This approach produces an efficient set of test vectors, able to reach many branches as compared to earlier approaches such as HYBRO [26]. However, due to design complexity scaling up, some of the hard-to-reach branches remain difficult to reach.

To improve the utility of BEACON and to tackle the aforementioned problems, we present a new methodology which uses two different coverage metrics in two different stages. In the first step, test vectors are generated using a newly proposed technique with Mutation Coverage as the target. In the second step, these mutation vectors are used as a seed to the BEACON tool, resulting in a set of final test vectors with an improved branch coverage. This combined approach can help cover the hard-to-reach branches, resulting in an improved branch coverage with smaller numbers of test vectors as compared to the original BEACON. However, this comes with a cost in test generation time. To reduce execution time, we also present techniques by reducing the number of mutations of this new methodology.

1.2 Contributions of the Thesis

The contributions of this thesis can be seen as follows.

1.2.1 Dual Metric Test Generation Framework

We introduce a dual metric test generation methodology, which aims to generate test vectors that maximize the branch coverage of an RTL design, using the aid of an additional coverage metric called mutation coverage. Moreover, this methodology was designed with a goal of

improving the performance of the existing technique (BEACON) with respect to the branch coverage.

Mutation Coverage

Mutation testing [8], also known as Fault-based injection testing at the hardware level, has gained popularity during the past couple of decades as an effective coverage metric for Verilog and SystemC simulations [31],[25],[30]. In Mutation Testing at the RTL, an error is purposefully introduced into the design. For example: arithmetic operator '+' replaced by '*'. If there is a difference in the output of the designs after a stimulus is applied, then we say that mutant is killed. Mutation coverage is the ratio of number of killed mutants to that of total mutants simulated.

Now a question arises: *Why is mutation coverage chosen as a coverage metric to improve BEACON?* Experiments were conducted by introducing simple mutations into the design. It was observed that upon an application of directed test suites which kills many of these mutants, the directed tests helped change the direction of the flow in the golden design, thereby increasing the probability of uncovered branches getting covered. Effectively, killing some of the mutants may require a more diverse set of values to be experienced by internal variables, which result in traversing through different execution paths. Some of these paths may help to reach previous hard-to-reach sections of the design.

Different methods for test generation targeting mutants such as symbolic execution, constraint-based, and search-based have been extensively studied for software designs [34][7][22], and some of them have been even analyzed and studied for Verilog/VHDL Designs [14][5]. These generators aimed at generating tests for each individual mutant. Since our aim of having mutation coverage test generator is primarily to improve BEACON, our methodology for

mutation coverage follows the same search-based strategy used in BEACON, as it scales to large designs better than formal methods.

Framework Overview

The proposed framework is composed of two stages. The first stage generates test vectors which aim at maximizing the mutation coverage of the given RTL. These vectors are used as an initial population to the original BEACON in the second stage. The output of the second stage is the final set of test vectors which maximizes the branch coverage. Following is a short overview of the working of the first stage of our framework:

Firstly, HDL is converted to C++ using Verilator [32]. Instrumentation for the converted HDL is done at compile time. Verilator is modified to generate mutants during conversion. A mutant is applied to the golden C++ model to create a mutated version of the same. Instrumentation is added to both versions (golden & mutated) of the design, for each branch having variables dependent on the mutant. Also, the updated values of such variables are tracked in a special data structure for each instrumented branch. Hence the number of actually instrumented branches also otherwise named "Mutation_Branch", will be less than or equal to the total number of branches. Now the Ant Colony Optimization (ACO) based search begins when each ant is initiated by random vectors and this same set of vectors are applied on both versions of the model in parallel. The instrumented branches with the variables are recorded. The variables for the corresponding branches in the golden and mutated C++ versions are compared and if there is a difference, it implies the mutation has started propagating and the corresponding branch number is marked visited. The score generated by the fitness function for each mutation_branch is proportional to the inverse of the frequency of that branch being visited. This fitness score helps the tool narrow down the search space on the non-visited mutation coverage branches, i.e., it tries to maximize the

effect of mutation in the design. The above procedure is repeated until either the output variables differ in their golden and mutated version or a maximum number of iterations has reached. If the former is true, we can say that mutant is killed and the test vectors are saved. The time for compilation to C++, generation of mutants and the instrumentation of mutation coverage branches is very less as compared to the overall cost, and can be considered as a one-time preprocessing cost.

The dual metric methodology generated high-branch coverage tests for hardware benchmarks from ITC99 [6]. Experiments show that our methodology were able to improve the branch coverage of some of the benchmarks with also a decrease in the number of test vectors. Though for most of the circuits, execution time decreased, but yet for some circuits, the overall test generation time increased due to the extra time consumed by the first stage in analyzing all the generated mutants.

1.2.2 Mutant Filtering

To address the increase in test generation time as discussed in the previous section, the next step in this thesis was to optimize the first stage. We are aware of the fact that there are infinite mutations possible for any RTL, comprising of different types like relational, arithmetic, conditional, variable, constants. Within this infinite mutant space, there are several such mutants which may be redundant for improving the branch coverage. Hence, costing unnecessary simulation time on generating tests to kill such mutants doesn't serve any purpose. Methodologies have been proposed in the past years by the researchers to identify useful mutants within the software testing [34], but the research at hardware level has not scaled accordingly. Hence, in this part of the thesis, we present our following contributions:

- We introduce two mutant filtering approaches to optimize the mutant space for our

hardware testing. Useful mutants are identified using the information extracted from the Data Dependency Graph (DDG) and Control Flow Graph (CFG) of the given RTL. Tests generated for the reduced set of mutants shows an improvement in the execution time when compared with those generated by a larger mutant space, with an equal number of covered branches.

- We study these approaches in detail and present some analysis on the nature of the useful mutants and their dependency on the structure of the RTL, by evaluating the effort involved in the improvement of branch coverage for some of the ITC 99 benchmarks [6]

1.3 Thesis Organization

Rest of the thesis is organized as follows.

- Chapter 2 describes the background concepts used in explaining this work.
- Chapter 3 describes the contribution, experimentation and results of the first part of this thesis in detail
- Chapter 4 describes the contribution, experimentation and results of the second part of this thesis in detail
- Chapter 5 describes the work carried out at Tesla related to this thesis
- Chapter 6 concludes the work with a summary of findings and possible future directions for the research.

Chapter 2

Background

In this chapter, we present the key terminologies and fundamental concepts used in our work, which will help to give a more detailed understanding of the problem. We also present a study on some previous methodologies proposed by researchers in the area of RTL (Register Transfer Level Description) test generation.

2.1 Genetic Algorithm

Genetic algorithm (GA), first proposed by Holland [15] is a meta-heuristic motivated by the process of natural selection. They use bio-inspired operators such as mutation, crossover, and selection to generate high-quality solutions for search problems. It is better than Artificial Intelligence (AI) in terms of robustness, where they do not break easily if the inputs changed slightly or in the presence of a considerable noise. In GAs, there is a pool or a *population* of possible solutions to the given problem. Then a repeated process of *recombination* and *mutation* (just like in DNA) over various generations undergoes, producing new children after each process. Each candidate solution (individual) is assigned a fitness score and the ones with higher fitness are given a higher chance to mutate and gradually evolving the population towards an optimal one. The GA process is divided into three phases as shown in Figure 2.1. The first is the Fitness Evaluation phase, where the fitness is evaluated for the initial population of individuals to find the fittest ones. The next phase is the *Selection*

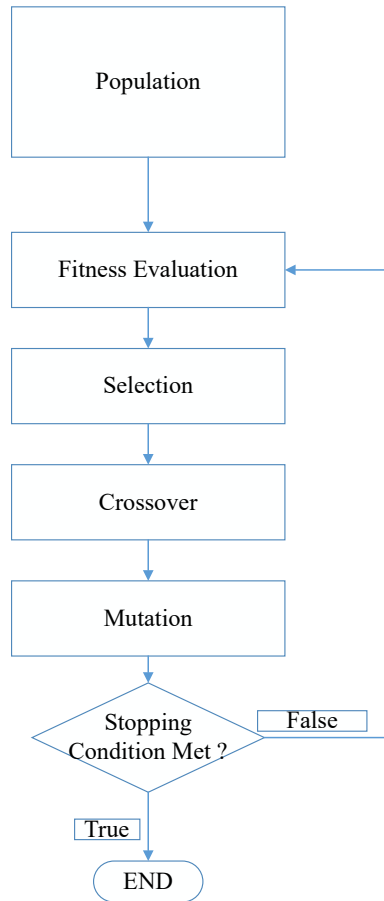


Figure 2.1: Phases in Genetic Algorithms

phase, where the idea is selecting the fittest pair of individuals (parents) from the current population and pass their genes to the next generation. The next phase is the *Crossover* phase, where the selected pair of the parent is mated i.e. genes of the selected pair of parent are exchanged among themselves up to a random-chosen crossover point resulting in a pair of new offspring (children). To maintain diversity and prevention of premature convergence, we have the next phase called *Mutation* phase, where new genetic material like the flipping of bits are introduced. But also, this phase is less frequent as [24] stated, high mutation can lead to an undesirable effect on the algorithm convergence. This cycle is repeated until the stopping criteria are not met. GAs have been successfully applied to test generation in the past, most notably at the gate-level [16, 17, 19, 23].

2.2 Ant Colony Optimization

”The ACO algorithm [11] is another biologically inspired algorithm, whose aim is to convert the problem into a search problem between an ant colony, or nest, and food source(s). Using local pheromone trails for information exchange, the process of ants’ reinforcement learning can be formulated as a meta-heuristic algorithm to solve NP-hard search problems”[24]. The basic idea of the ACO algorithm is formulated as the process of ant foraging. The problem is first mapped to a graph, in which edges between nodes in the graph denote paths and trails that ants can traverse. Initially, from the starting node, ants wander randomly through the paths(edges) of graph G. After an ant finds a solution (food), based on its attractiveness (cost), the ant will lay down an appropriate amount of pheromone along the trails (edges) it has traveled. If any other ant finds such a path, they will not likely wander randomly, but instead follow the trail, returning and reinforcing it, if they eventually find the solution. This process is called reinforcement. Over time, the pheromone trail starts evaporating and worsens for the longer path since the time taken by the ant to trail will be higher. This process of reduction in its attractiveness is called evaporation. With this process of reinforcement and evaporation, ants end up trailing the shortest path with high pheromone density. In other words, when an ant finds an optimal(short) path to traverse, all other ants follow the same eventually reaching the target state.

ACO has shown to be a very effective approach towards many domains of NP-hard optimization problems such as in the Telecommunication Networks routing problem, Travelling salesman problem [10], Industrial problems like scheduling problem etc.

However, test vectors generated by either genetic algorithms or ACO tend to be very long. Although methods to compact the vectors exist [18, 28], they incur additional computational cost. Therefore, it would be beneficial to generate vectors that are short to start

with.

2.2.1 Mutants

Mutation testing and analysis have been a popular testing method in the software testing method for decades [13]. They have also gained popularity among researchers for RTL functional testing in the past few years [5, 14] due to their close resemblance with the Software testing. It is a fault-based testing method, in which an error or a fault is deliberately introduced into the design and we check if the tests generated are able to detect the fault. The error introduced is called *mutant* and these mutants are represented as syntactical changes to the design. Syntactical changes include changes in operators, values of constants, deletion of certain statements etc. For example: a '+' *add arithmetic* operator replaced by a '*' *multiply*:

$$\text{golden: } P \leq Q + R; \text{ Mutated: } P \leq Q * R;$$

There is an infinite set of possible mutants in a design. In our proposed approach, our mutant generator generates mutants limited to only Arithmetic Operators, Logical Operators, Relational Operators and values of constants. The aim is to create a test suite which can differentiate each mutated RTL from the golden RTL, by causing the mutant to produce different outputs. A mutant is termed as a *killed mutant* if such differentiation is possible with some input to the design. Generation of such test cases and analyzing them as above, allows the mutation analysis to judge the quality of tests through a metric called mutation score or mutation coverage. The mutation coverage "MC" relatively to a test suite "TS" and RTL design 'R' is computed as below:

$$MC(TS, R) = 100 * \frac{K}{T} \quad (2.1)$$

where T is the total number of generated mutants, and K is the number of killed mutants.

```

1 always @ (posedge
   clock)
2 begin
3   if (reset) begin
4     state = 2'b00;
5   end else
6     case(state)
7       2'd0 :
8         state = 2'd1;
9       2'd1 :
10        state = 2'd2;
11       2'd2 :
12        begin
13         if(counter >= 2)
14           state = 2'd0;
15         else
16           state = 2'd2;
17        end
18       default : state = 2'
19         d0;
20     endcase

```

Listing (2.1) (a)

(a) Verilog HDL

```

1   if (reset) {
2     ++(Vcoverage[0]);
3   } else {
4     ++(Vcoverage[9]);
5     if ((0U == (IData)(state))) {
6       ++(Vcoverage[1]);
7       state = 1U;
8     } else {
9       if ((1U == (IData)(state))) {
10        ++(Vcoverage[4]);
11      } else {
12        if ((2U == (IData)(state))) {
13          ++(Vcoverage[7]);
14          if ((2U <= (IData)(counter))) {
15            ++(Vcoverage[5]);
16          }
17        }
18      }
19    }
20  }

```

Listing (2.2) (b)

(b) Verilated C++ Model

Figure 2.2: Verilog HDL Snippet and its corresponding C++ verilated model

2.3 RTL Instrumentation

Verilator [32] developed by Wilson Snyder, with Duane Galbi and Paul Wasson, is an open-source and a free software tool which gained popularity in the recent times in the field of RTL Verification. It compiles a synthesizable Verilog (RTL) into a cycle-accurate behavioral C++ model. Complicated and huge designs where fast simulation performance is the main concern, Verilator comes to rescue. All signals (IO and internal) are represented as the public members of this C++ model, accessible by any software model using this class. It

provides options to unroll a loop, different levels of code optimization, code instrumentation, etc. The class also contains an important main function *eval*, which provides the ability to simulate the behavioral response of the design with the updated values of the signals. For example, in order to simulate a part of the design dependent on the rising edge of the clock, the model can be simulated as assigning data inputs followed by two calls to *eval* with the clock input *clk* toggled for each call. Verilator also helps to instrument the generated C++

```

132c132
<          vITOPp->v_DOT_counter=vITOPp->v_DOT_counter + 1U;
-----
>          vITOPp->v_DOT_counter=vITOPp->v_DOT_counter - 1U;

```

Figure 2.3: Example Patch file for a sample mutant

model based on the branches, which plays a vital role in determining branch coverage of the design. This is represented by the *Coverage* function generated in C++ model. The Branch coverage tracker is a contiguous array *VCoverage* used within this function with the length equal to number of branches in the given RTL design. For each branch exercised, its corresponding array element is incremented. For every RTL to C++ conversion, Verilator creates a main module (top class) *Vtop.cpp*, which instantiates other relevant classes and functions including *VCoverage* and *eval*. A wrapper C++ file is written to simulate the C++ model by instantiating the class *Vtop* and initialize its members. An example of an RTL snippet and its corresponding C++ verilated model is shown in Figure 2.2. Verilator is an integral part of our work, mainly used in the Preprocessing phase of our methodology as presented in detail in Chapter III. For our work, Verilator is modified to enable generation of mutants in the form of patch files, while converting RTL to C++. An example patch file is shown in Fig 2.3. This process of generating mutants saves a huge cost of generating mutants with respect to time. These patch files are applied to the original RTL and mutated versions are generated. Additionally, Control Flow Graph and Data Dependency Graph are

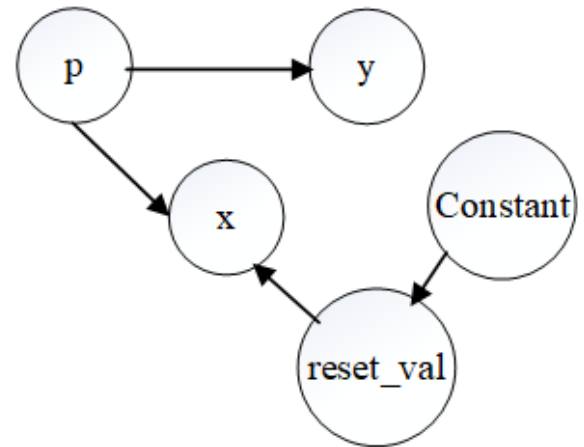
extracted with the help of the Abstract Syntax Tree generated by the Verilator.

```

1  if( flag )
2      branch_pt[0]++;
3      reset_val = 5;
4      p = 1;
5  else
6      branch_pt[1]++;
7      y = p - 1;
8  if( reset_val <= 15)
9      branch_pt[2]++;
10     reset_val = 25;
11     if(p == 1)
12         branch_pt[3]++;
13         x = reset_val + p;
14     else if (y == 0)
15         branch_pt[4]++;
16 else
17     branch_pt[5]++;
18     x = 2;
19     if(y == 0)
20         branch_pt[6]++;
21         reset_val = 0;
22     else
23         branch_pt[7]++;
24         reset_val = 15;
25         y = x + p;

```

(a) verilated RTL



(b) DDG

Figure 2.4: Example of verilated RTL with its DDG

2.4 Data Dependency Graph

A data dependency graph (DDG) is a directed graph, generated for a program, representing in the form of vertices(V) and edges(e). Each vertex represents either a primary input to the program or an intermediate value being used in the program or a constant. The edges of the graph represent the dependencies between the vertices. The vertex which has an incoming

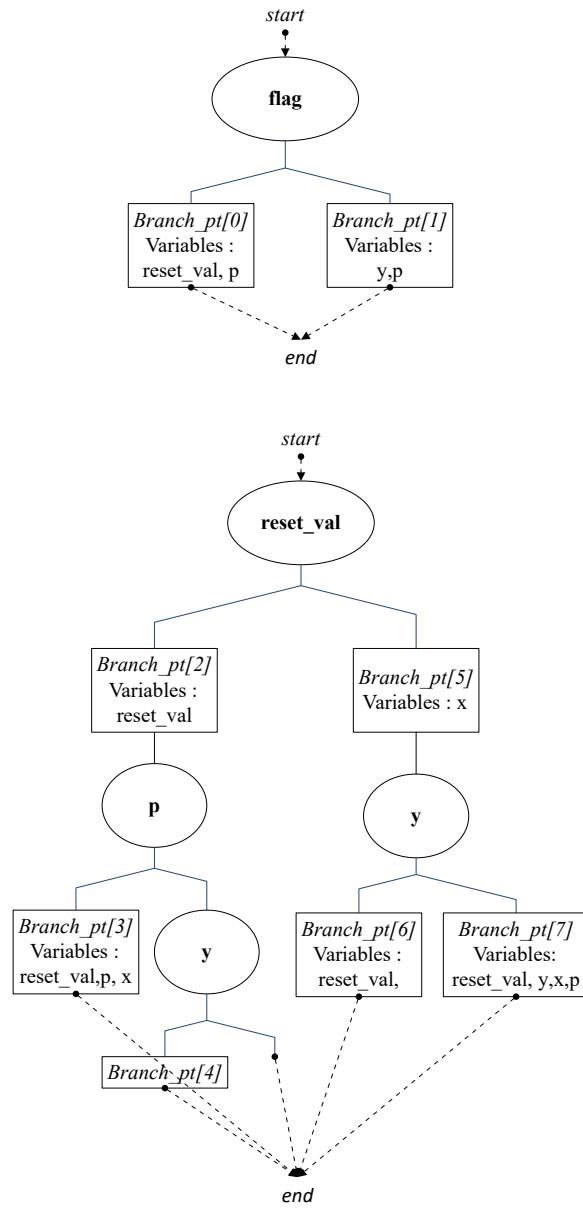


Figure 2.5: CFG for the above verilated RTL

edge implies that this vertex is statically dependent on the vertex originating from this edge. Fig 2.4 shows an example of a DDG of a sample RTL.

2.5 Control Flow Graph

Control Flow Graph (CFG) first proposed by Allen [2], is also a directed graph, representing the flow of control during the execution of a program. Nodes in this graph represent basic blocks and edges represent the control flow paths between these basic blocks. A basic block is a set of instructions executing sequentially and having only one entry point and one exit point. In software, usually, CFG is in the form of a tree as there are single entry and exit points in the overall program. But in hardware, we have multiple paths from input to output since hardware simulations are cycle based and the design runs in parallel. Hence, CFG in hardware is extracted per cycle instead of all cycles together. They are composed of multiple trees unlike in software. For example, in Figure 2.4, We have two hierarchy of trees generated of the CFG (Fig 2.5) for the sample RTL. The elliptical shape box represents the conditional variables. The rectangular box represents the basic block with the corresponding instrumented branch counters. The *Variables* within the basic block gives all the relevant signals/registers being used in those block of instructions.

2.6 Related Work

In BEACON [24], a bio-inspired meta-heuristic demonstrates viable RTL test generation by using the branch coverage as a metric and a self-refining Evolutionary Ant Colony Optimization(ACO). It is a simulation-based swarm intelligence technique with random vectors as the initial seed to the population, helping explore the circuit over a greater number of cycles,

and avoiding the computational cost of excessive SMT solver calls as observed in HYBRO. Although BEACON can help achieve a high level of branch coverage with a capability to also cover many hard-to-reach branches, yet there were a lot of ITC99 benchmarks like b06, b07, b11, b12 where some of them remain uncovered or it required a huge number of cycles to reach them. Since the initial seed is random vectors, the exploration of the design starts randomly and then the guidance takes into effect. The final state of BEACON leaves the design explored to its maximum capability. Hence, for some of the designs, some hard-to-reach branches could not be covered, which is a limitation of BEACON and motivation for the proposed technique.

PACOST [36], a target oriented RTL test generator has shown success in reaching hard-to-reach states operating in an abstraction-based semi-formal verification framework. It uses factored explorations added to interleaved concrete and symbolic simulation of the design. But this tool is not scalable to large designs due to the limitations faced by formal verification techniques and the inconsistency of the generated abstract models. SMT solvers have been used extensively in HYBRO and PACOST. Each SMT instance essentially encodes a specific execution path symbolically, which the SMT solver attempts to satisfy. If satisfiable, it also generates the corresponding stimuli that exercises the execution path. One can also encode the instance as a model checking or bounded model checking instance, either with SMT or SAT formulation. Incremental solving by adding constraints have shown to improve the solver performance [3, 35].

Researchers in [5, 7, 14, 22, 34] proposed various methodologies to generate tests for high mutation coverage in software and hardware designs. These approaches included symbolic based execution, constraint based simulation. All these approaches demonstrated a high mutation coverage for their corresponding set of mutants. Since the complete mutant space is infinite, these approaches and the proposed approach cannot be directly compared unless

the identical sets of mutants are used.

Compared to the above-mentioned approaches including the abstraction-based and symbolic execution based, the Mutation coverage branch that we extract from a design, as we present in the next section, represent the static structure of the circuit. No symbolic methods are required for the method, and we resort only to pheromones from the actual simulation for guidance in our search.

Chapter 3

Dual Metric RTL Test Generation

In this chapter, we present a new methodology generating test vectors for the RTL design. This methodology increases the branch coverage by covering the hard-to-reach branches which couldn't be covered by BEACON. We also present the experiments and results to support our claim.

3.1 Framework & Methodology

The high level framework divided into multiple stages is shown in Figure 3.2, 3.5 and 3.6. The user is expected to provide the Verilog Hardware Description Language (HDL) source for the design. The user also initially configures the tool with several options including the number of cycles for the tool to run, number of repetitions, to be used for test generation.

We first explain the reason behind the use of mutation test to help achieve high branch coverage. Consider the example in Figure 3.1. We have a snippet of a sample design, a branch statement (*if* statement on line 14) which contains two control variables a and b , related by a relational operator '>'. By conventional test generation that targets this branch, the condition that "a greater than b" must be satisfied. Now say suppose, we mutate any statement within the branch (line 16 or 17) and try to kill the mutant by finding the appropriate test vector. This process of finding test vector, is a part of mutation coverage based test generation. It implies that this test vector that ensure the detection

```
1.  module DUT (.....)
2.  reg a, b , X, t, y;
3.  ....
4.  ....
5.  ....
6.  ....
7.  X = t + y;
8.  ...
9.  a = X * t;
10. b = 2y;
11. ...
12. ...
13. always @(posedge clk)
14. if(a > b)
15. begin
16. z = y ^ 2;
17. t = t + 2;
18. else
19. $display("We are accessing this branch" );
20. endmodule
```

Figure 3.1: Example Verilog to Justify Methodology

of the mutated statement will also exercise the condition on line 16. When this test is embedded in the initial seed to BEACON, the search will be biased toward the statement which was mutated. Hence, choosing mutation coverage as the metric is an appropriate choice for helping reach hard-to-reach branches and improving BEACON.

Furthermore, suppose that we do not find any test vector which could kill mutants generated from line 16 and 17 in the design. Now in order to satisfy the conditional statement on line 14, we must set the right values to the control variables a and b . If we mutate line 7, the

new value of X may modify the value of a such that the conditional statement is satisfied. So we can see that, mutants outside the uncovered branch space can also help achieve the expected branch coverage.

The test generator is divided into two stages. The first stage generates the test vectors based on mutation coverage metric. It starts with a preprocessor generating a cycle-accurate functional simulator from the given Verilog alongside extracting mutants and other various design information [29]. Then, the test generation stage receives two versions of the design for each mutant, i.e., golden and a mutated version. Evolutionary Algorithms with Ant Colony Optimization based search is applied with an aim to guide the propagation of mutant to the output. The resulting output of this stage is the set of test vectors which can achieve high mutation coverage. The second stage seeds BEACON with the generated test vectors followed by random input vectors. Once a fixed number of iterations is reached, the input vectors that form the optimal test suite for improved branch coverage are written out to a file. A detailed description of each stage is presented below in the following subsections.

3.1.1 Preprocessor

The Preprocessor begins by translating the source (Verilog) of the given design into C++ using Verilator. A brief overview of the Pre-processing phase is shown in Fig.3.2. Verilator is an open source software tool that, compiles a synthesizable Verilog into a cycle-accurate behavioral C++ model. All signals (IO and internal) are represented as the public members of this C++ model, accessible by any software model using this class. It provides options to unroll a loop, different levels of code optimization, code instrumentation, etc. The class also contains an important main function *eval*, which provides the ability to simulate the behavioral response of the design with the updated values of the signals. For example, in

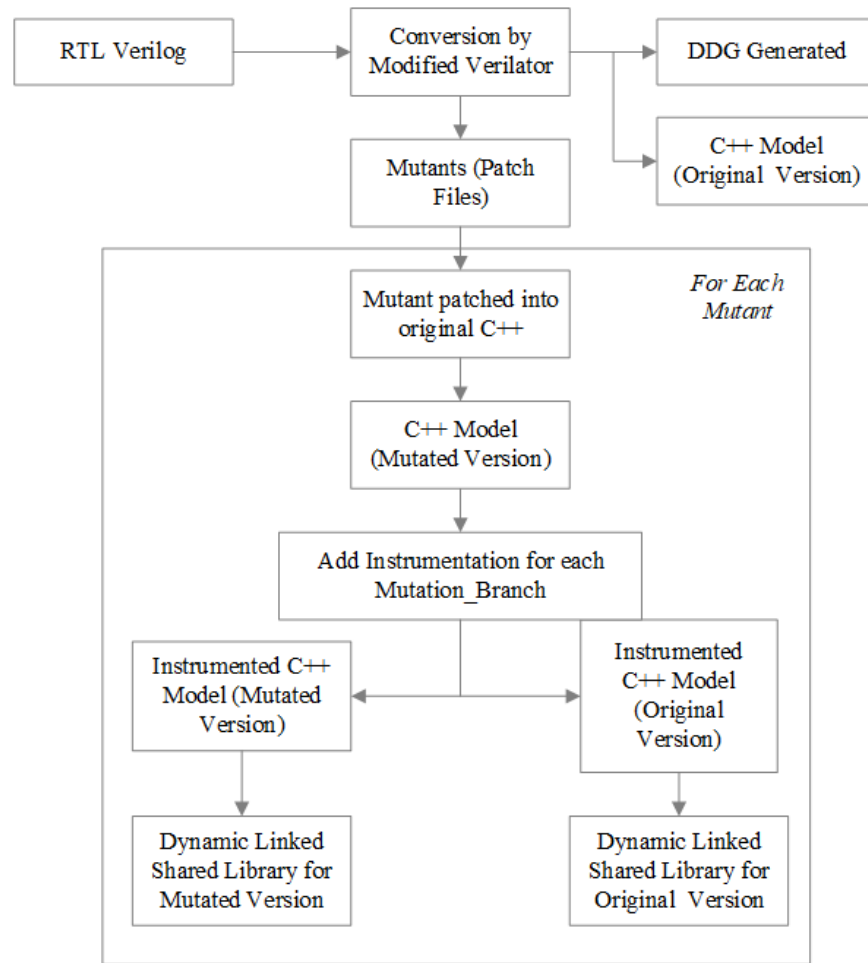


Figure 3.2: Preprocessing for Mutation Coverage Guidance Search

order to simulate a part of the design dependent on `@posedge clk`, the model can be simulated as assigning data inputs followed by two calls to `eval` with the clock input `clk` toggled for each call. Verilator also helps to instrument the generated C++ model based on the branches, which plays a vital role in determining branch coverage of the design. This is represented by the `Coverage` function generated in C++ model. Listing 2.1 in Fig. 3.4 shows a sample Verilog description of a design to showcase the usage of Verilator.

```
132c132
<          vITOPp->v_DOT_counter=vITOPp->v_DOT_counter + 1U;
-----
>          vITOPp->v_DOT_counter=vITOPp->v_DOT_counter - 1U;
```

Figure 3.3: Example Patch file for a sample mutant

For our work, we modified the Verilator source code to generate mutants during the conversion process. Every single C++ assignment statement, while being outputted from Verilator, is checked whether it contains any operators belonging to any of the categories: arithmetic, logical or relational or if it contains a constant value. If present, then the operator/values of interest are replaced with operators/values with the same category. Then, a patch file (.patch) is generated as shown in Fig. 3.3 which contains the two versions, i.e., golden and mutated. Hence, we generate all possible such mutant files with a negligible overhead to the overall cost.

```

1 always @ (posedge clock)
2 begin
3 if (reset) begin
4 state = 2'b00;
5 end else
6 case(state)
7 2'd0 :
8 state = 2'd1;
9 2'd1 :
10 state = 2'd2;
11 2'd2 :
12 begin
13 if(counter >= 2)
14 state = 2'd0;
15 else
16 state = 2'd2;
17 end
18 default : state = 2'd0;
19 endcase

```

Listing (3.1) (a)

(a) Verilog HDL

```

1 if (reset) {
2 ++(Vcoverage[0]);
3 counter = 0U;
4 (monitor_mutants).push_back(
5     uint32_t(3));
6 (monitor_mutants).push_back(
7     uint32_t(0));
8 (monitor_mutants).push_back(
9     uint32_t(0));
10 (monitor_mutants).push_back(
11     uint32_t(counter));
12 ++(mutation_coverage_id[uint32_t(0)
13     ]);
14 } else {
15 ++(Vcoverage[9]);
16 if ((0U == (IData)(state))) {
17 ++(Vcoverage[1]);
18 state = 1U;
19 (monitor_mutants).push_back(
20     uint32_t(4));
21 (monitor_mutants).push_back(
22     uint32_t(1));
23 (monitor_mutants).push_back(
24     uint32_t(1));
25 (monitor_mutants).push_back(
26     uint32_t(counter));
27 (monitor_mutants).push_back(
28     uint32_t(state));
29 ++(mutation_coverage_id[uint32_t(1)
30     ]);
31 } else {
32 if ((1U == (IData)(state))) {
33 ++(Vcoverage[4]);
34 } else {
35 if ((2U == (IData)(state))) {
36 ++(Vcoverage[7]);
37 } if ((2U <= (IData)(counter))) {
38 ++(Vcoverage[5]);
39 state = 0U;

```

Listing (3.2) (b)

(b) Verilated C++ Model

Figure 3.4: Example Verilog HDL Design and Instrumented C++ Model with Mutation Coverage and Branch Coverage Counters

Along with mutants, a simple harness that instantiates the C++ model and provides generic accessor functions to its public members (signal variables, counters tracking the coverage, reset, clocks and *eval*) is also generated. Verilator provides an option of dumping an Abstract Syntax Tree (AST) for a design considered for conversion. It gives all the information about each and every statement in the design. We parse the tree to create DDG. In our work, DDG is generated for RTL and since it is a cycle based execution, unlike for software program, the data dependency is not only from one variable to another but also exists across cycles. Whenever the mutated statement gets executed, the left-hand side value of the assignment statement is expected to differ from that of golden sim and the difference propagates to the output. So, it is very important to understand what variables are dependent on this changed value, to help correctly guide the test generation process. Using the DDG, we can extract all the variables which are dependent on the left-hand side variable of an assignment statement.

Next, the patch file containing the mutant is applied to golden C++ model to create a mutated version of C++ model. Then we parse the two versions of the models and locate all the branches containing any of the extracted dependent variables affected by the corresponding mutant. These branches can be called as a *Mutation_branch* for future reference. Each *Mutation_branch* is then instrumented with a new counter *mutation_coverage* and also a globally declared data structure to track the updated values of these relevant variables. Here, *mutation_coverage* is globally declared as an array with the number of elements equal to the number of *Mutation_branches*. An example of the above-mentioned instrumentation for the reference design of Listing 3.1 in Fig. 3.4 is shown in Listing 3.2. Also, the harness file is modified to accommodate the accessibility of *Mutation_branches*. Finally, the preprocessor phase ends with the compiling of the two versions of C++ model to the corresponding dynamic libraries, which will be each linked with the proposed tool for design validation using a self-written Makefile. The above process is repeated for all the generated patch files.

3.1.2 First Pass Test Generation(TG)

The First Pass includes a test generation engine, implemented as an ACO meta-heuristic search technique with random vectors as the initial seed. The framework is shown in Fig. 3.5.

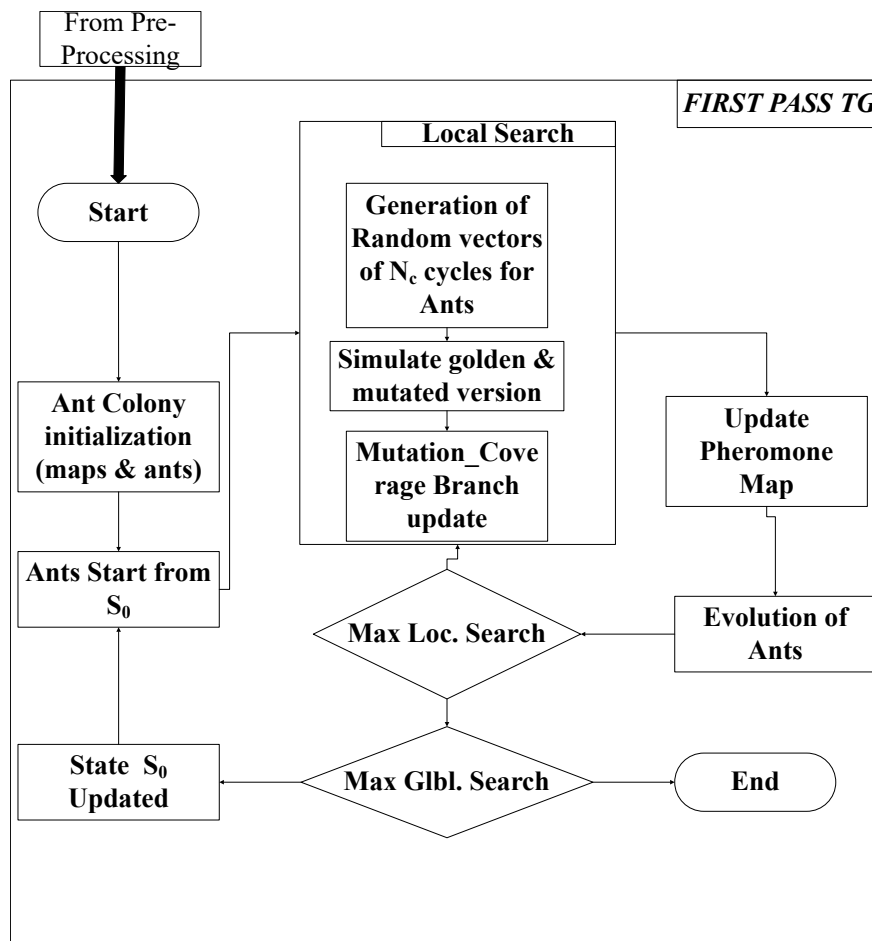


Figure 3.5: First Stage of Dual Metric Test Generation

The process is similar to the one used in BEACON with a difference in the search guidance method and focuses on a different metric i.e. Mutation coverage. The tool, unlike BEACON, performs two parallel evaluations on a pair of design versions (mutated and golden), which are

instrumented with both `mutation_coverage` and `branch_coverage` information as shown in Figure 2.2. As shown in Figure 5, the guidance framework starts with the initialization of the pheromone map (ϕ) based on the database of branches which contains `mutation_coverage` counters aka *Mutation_branch*. We have K ants, each initialized with a randomly generated vector sequence of N_c cycles. The same set of ants and vector sequences are applied to both the versions of the RTL C++ model instances. After initialization, the First Pass search proceeds in the following manner: The K ants start from initial state S_0 . For each iteration t , ant_k following the vector sequence with length of N_c cycles is evaluated using *eval()* function provided by Verilator. For each Mutation branch, which the path traverses, the updated values of signals/variables of interest (ones extracted from DDG, dependent on the mutated statement) is recorded for both the parallel simulations. After N_c cycles for ant_k , if the final recorded values per mutation branch are not equal in the golden and mutated version, then the corresponding `mutation_coverage` counter is incremented. On similar lines of the Local Search Algorithm in BEACON, if a new `mutation_branch` is covered, the current state of the design is saved to *saved_set*, which will be used as a starting state in next iteration. Once all the ants finish the simulation, pheromone map is updated via a process of reinforcement and evaporation. Then ants generation is evolved based on the fitness calculated based on the number of pheromones. This search termed as the Local search terminates when no more new `mutation_branches` are discovered for R rounds or the `mutation_branch` containing output variables is hit. If the former is true, then the state saved in the present local search is set as the initial state for the next round of global search, otherwise, even the global search terminates and we can say that the current mutant is killed and vectors are saved. If the number of rounds for the global search reaches maximum (user defined), then this mutant is said to be a non-killed mutant. Following below are the stages of the search guidance:

Pheromone Update

Pheromone map is updated in two stages: Reinforcement and Evaporation. After each ant finishes N_c cycles of simulation, mutation_branch trace mb_trace^k , is recorded and the pheromone gets deposited proportional to the number of times the specific mutation_branch is visited, similar to that in BEACON. Hence, more the pheromone accumulation implies that the branch is easy to reach. But, there is an additional factor which is considered here the amount of pheromone updated for the mutation_branches which contains the output variable needs to be lesser than the other branches. Hence, by doing this, the output variable branches would be considered to be a hard-to-reach. It is shown in Equation 3.1

$\forall mb \in mb_trace^k :$

$$\begin{aligned} \phi_{mb}(t+1) = & (\phi_{mb} + \frac{N_m}{N_c} \times P)(1 - B_o)(1 - B_I) \\ & + (\phi_{mb} + P)(B_o + B_I) \end{aligned} \quad (3.1)$$

where ϕ_{mb} is the pheromone amount deposited at time t on Mutation_branch mb . B_o and B_I represents whether the branch being monitored is the output variable containing mutation_branch and mutated statement containing mutation_branch respectively and P represents the pheromone amount to be deposited (user defined). In regards, to the evaporation process, the pheromone also evaporates gradually as the time taken to traverse by the ant progresses, causing extreme pheromone degradation for the hard-to-reach branches. This will make the ants focus on these paths and according to the fitness function in Equation 3.2, mutation_branches with less pheromone will help achieve the goal.

$$fitness(I_c^k) = \sum_{mb \in MB_c^k} \frac{1}{\phi_{mb}} \quad (3.2)$$

where the above fitness calculation at a State T traversed in cycle count c by ant_k is the sum of reciprocals of the pheromone amounts on `mutation_branches` in the overall set of `mutation_branch` set MB_c^k .

Ants Evolution

The evolution process is similar to as described in [24], which enables selection of good vector sequences and prune the vectors which divert the focus of simulator from the target `mutation_branch` (branch containing output variable).

3.1.3 Second Pass Test Generation

Test vectors generated from the First Pass engine, as discussed above focused on the reachability and observability of the mutated statement, with an aim to kill the mutant. These test vectors aka *Mutation test vectors* (reference name for future) are provided as the initial seed (initial state) to the existing BEACON. BEACON starts the search by evaluating the design using these test vectors, and since these vectors ensure that the corresponding mutants are killed, which implies that, these vectors will at least ensure the access of the mutated statements in the golden design, unlike the random vectors. Hence, the probability of accessing the part of design untouched by the golden BEACON increases. The framework for the Second Pass Test Generation engine (Figure 3.6) is the BEACON framework with an initial seed of random vectors replaced by the Mutation test vectors.

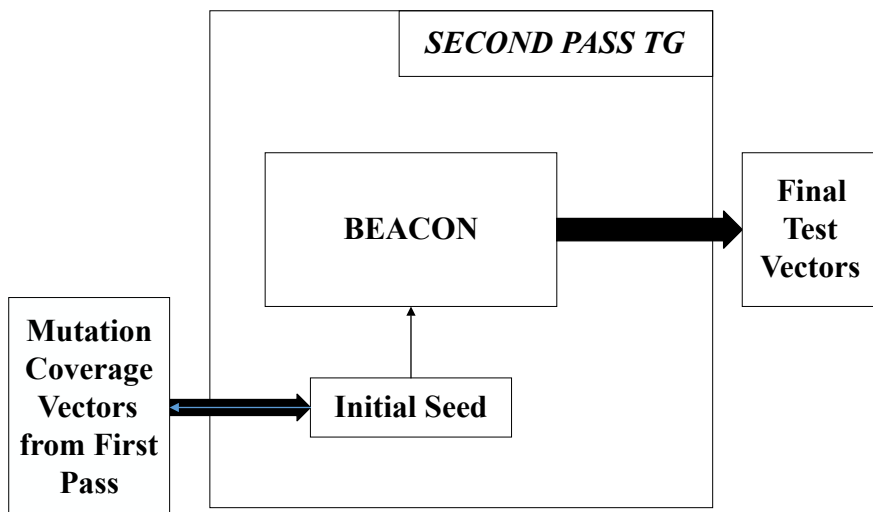


Figure 3.6: Second Stage of Dual Metric Test Generation

Experiments were conducted across multiple ITC99 circuits as discussed in Section IV and it was observed that, for certain circuits, we were able to cover additional branches which were missed by BEACON.

3.2 Experiments & Results

The Framework is developed in C++. All the experiments were conducted on a Linux(Kubuntu 4.13.3) machine with an Intel Core i7-6700K CPU @4GHz with 62.8 GiB of memory. We conducted experiments across benchmarks from ITC99[6] detailed in Table 4.2. We chose to

Benchmark	#Lines	#Branches	#PIs	#POs	#FFs
b06	128	24	2	6	9
b07	92	19	1	8	49
b10	167	32	11	6	17
b11	118	32	7	6	31
b14	509	211	32	54	245
b15	671	149	36	70	449

Table 3.1: ITC99 Benchmarks

Bench	Branch Coverage %		Pre.(s)	Exe Time(s)
	BEACON	Ours		
b06	95.83	95.83	0.03	0.262
b07	90	90	0.063	0.37
b10	93.75	96.88	0.078	4.22
b11	96.88	96.88	0.092	7.73
b14	91.94	94.7	1.47	163.85
b15	89.93	92.19	1.22	78.22

Pre: pre-processing time

Exe Time: similar for both BEACON and our approach due to same ACO configurations

Table 3.2: Branch Coverage

run our tool on the benchmarks of ITC99, which did not already reach 100 percent branch coverage by BEACON. The performance of our tool is presented in terms of the final branch coverage. The execution time is compared for both BEACON and our Dual Test generation tool.

3.2.1 Tool Settings

The mutants generated in the preprocessing phase belong to the category of arithmetic, logical, relational, constants, interchanged signal/variable names for our work. The number of mutants generated for each benchmark is shown in column 2 of Table 3.3. The Algorithm for the First Pass of our Test Generation has the following parameters set: the number of ants is set to $K = 120$ to enable greater communication between the ants. The maximum rounds (repetitions) for the Global Search is set to $R = 10$. The number of cycles (length of vectors) used for each round (Max rounds here is 3) of Local search are dependent on the size of the circuit. The Pheromone evaporation rate is set to 90%, the initial pheromone deposit is set to 1000 and maximum pheromone released by an ant is $P = 60$. The settings for the Second Pass Test Generation stage is similar to one used in BEACON experiments.

3.2.2 Branch Coverage

Our results are compared against BEACON [5] in Table 3.2. Column 3 represents the branch coverage for our new approach. The pre-processing time is shown in column 4 which is the one-time cost of generating mutants. The execution times for both the approaches, excluding the pre-processing time, shown in column 5 are given in seconds. We observe that the branch coverages are improved for certain benchmarks significantly. We also observe that the one time pre-processing cost is very less comparable to overall execution time of the tool, hence can be neglected.

Consider the circuit b14. BEACON achieved a branch coverage of 91.94% (with 196 branches covered out of 211 total branches). The circuit b14 is a subset of Viper processor, which contains large counters, the uncovered branches are dependent on its tendency to overflow. Our approach achieved 96.88% with the same execution time maintained as in BEACON. This major improvement in the branch coverage came about by applying a mutant. In another design, b15, it contains a 16-byte prefetch queue (array) in which opcodes and data are stored. Here BEACON achieved a branch coverage of 89.93% (with 127 branches covered out of total 141 branches). With our approach, we achieved a branch coverage of 93.6%, another significant improvement over the previous one with also notably less number of vectors (1145) than that generated by BEACON (38234). For circuit b07, our approach did not show improvement over BEACON in terms of branch coverage. This is due to the presence of branches which correspond to redundant else situations and default *case* statements, that were additionally inserted to make sure that sequential logic were correctly inferred by the synthesis tool. For other branches, we plan to identify more specific mutants which could help reach the uncovered branches and a way to automate them as our future work.

Bench	Total Mutants	Mutation Coverage %	
		BEACON	Ours
b06	151	98.01	100
b07	283	53.00	63.6
b10	720	80.9	80.9
b11	693	79.36	83.98
b14	2852	89.4	92.9
b15	3785	86.7	88.3

Table 3.3: Mutation Coverage

We note that the vectors generated during Stage 1 achieved a higher mutation coverage. Table 3.3 shows a comparison of Mutation Coverage achieved by BEACON and our Stage 1 vectors. Column 2 reports the number of mutants generated similar to in Table 4.2. Column 3 and 4 represents the mutation coverage for BEACON and our stage 1, respectively. We see that our new approach helps to kill more mutants than that by BEACON. This observation is justified by the fact that our approach updates pheromones in a way that the output variable containing branch is focused, rather than in BEACON which doesn't provide such specific guidance. Although the increase in mutation coverage is sometimes small, these might be the key mutants that lead to the coverage of additional uncovered branches in the design.

Chapter 4

Mutant Filtering

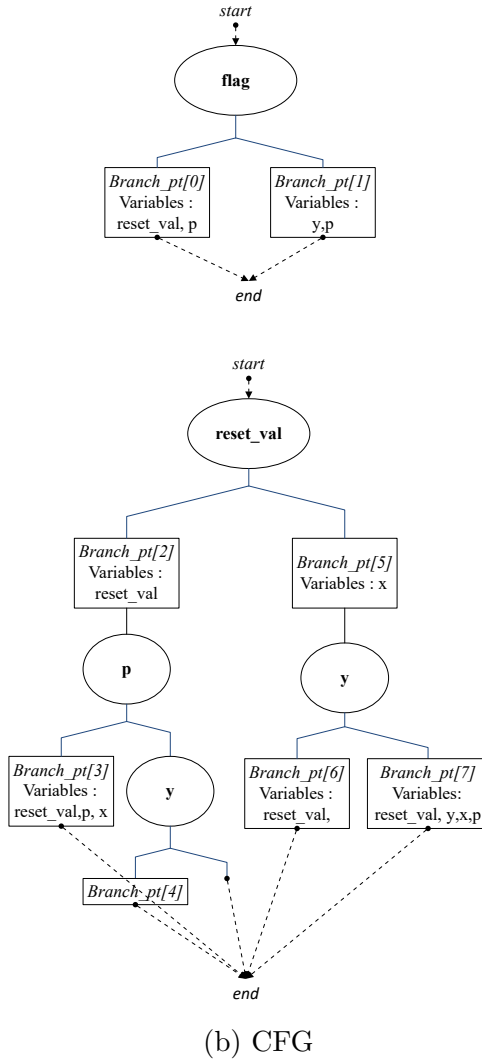
In this chapter, we present two mutant filtering techniques to improve the performance of our methodology presented in the previous chapter. We also present the relevant experiments and results show that test generation time and the test vector length show a reduction upto 85% and 80 % of those measured on BEACON. For rest of this chapter, let us take the following example of RTL and its CFG and DDG (Figure 4.1):

```

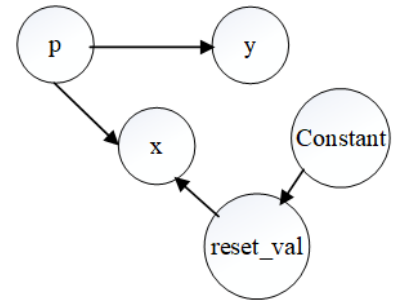
1  if( flag )
2      branch_pt[0]++;
3      reset_val = 5;
4      p = 1;
5  else
6      branch_pt[1]++;
7      y = p - 1;
8  if( reset_val <= 15)
9      branch_pt[2]++;
10     reset_val = 25;
11     if(p == 1)
12         branch_pt
13             [3]++;
14         x = reset_val
15             + p;
16     else if (y == 0)
17         branch_pt
18             [4]++;
19 else
20     branch_pt[5]++;
21     x = 2;
22     if(y == 0)
23         branch_pt
24             [6]++;
25         reset_val =
26             0;
27     else
28         branch_pt
29             [7]++;
30         reset_val =
31             15;
32         y = x + p;

```

(a) verilated RTL



(b) CFG



(c) DDG

Figure 4.1: Example of verilated RTL with its CFG and DDG

4.1 First Mutation Filtering Approach

In our previous and the original methodology as described in chapter 4 , we generate mutants for every statement in the RTL, resulting in a huge set of mutants. Then we generated tests to detect these mutants. Hence, a majority of the runtime is spent on the processing of all these mutants. Now, in any RTL design, there are two types of branches: a) Easily reachable b) hard-to-reach. If random test vectors are applied to the design for one round of simulation, experiments showed that some of the easily reachable branches can get covered. The runtime for this one round of simulation only depends on the size and complexity of the design and hence can be ignored when comparing with that of BEACON or our Dual Metric Test Generator. As a result, using mutants to cover these easily coverable branches is a mere waste of time and unnecessary.

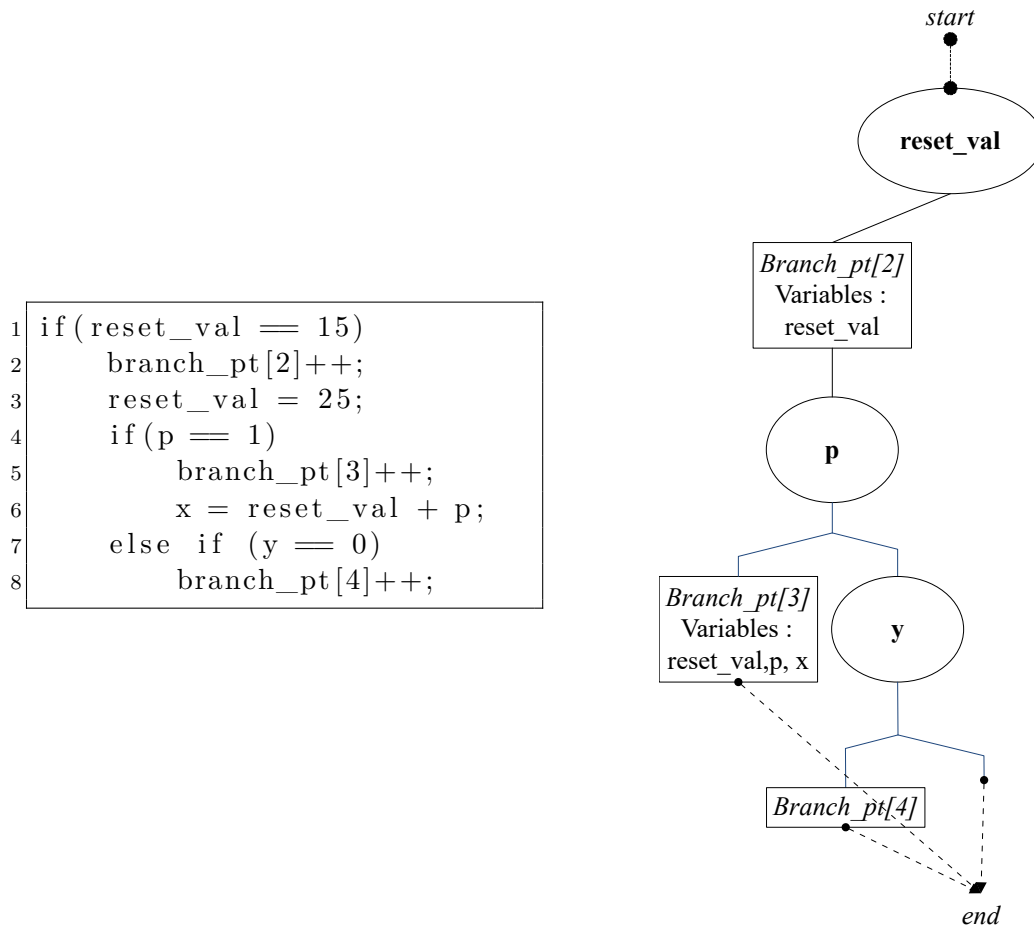


Figure 4.2: Snippet of Verilated RTL and its CFG

Following is a brief overview of our first mutation-filtering approach: The framework for this method is similar to that of our original framework, with an addition of another pre-processing step, where we simulate the verilated design with random test vectors and get the initial branch coverage information. Then, we extract all the control flow variables and their dependencies for each uncovered branch using the help of CFG and DDG. We generate mutants only for the conditional statements and statements in the basic blocks of CFG, which contain the extracted variables. This is because, to reach the uncovered branch, we need to satisfy its branch constraints and mutants for the same, can help achieve that.

For the example 4.1a, an excerpt of the verilated RTL and its control flow graph is shown

in 4.2 for an explanation. Say suppose after the initial round of simulation with random vectors: fourth branch (*branch_pt[4]*) is unreachable. Now as we observe from the Figure that the control variables which lie on the path from start to end through this branch are 'y', 'p', 'reset_val'. There is one basic block also present in this path. Hence, the possible mutants in this case which can help in covering this branch involve mutating the control flow statements at line numbers 1, 4 and 7, and basic block statements at line number 3. Next, using the DDG information, we find what statements affect the statements collected by the above control flow analysis. This way, we generate only relevant mutants corresponding to the uncovered branches unlike generating all the possible mutants.

After filtering the mutants, we follow the framework and procedure used in our original methodology. Using this approach, we reduced the mutant count by 50%, thereby reducing the test generation time by 40% for most of the ITC99 circuits. The details of the results will be discussed in Section IV.

4.2 Second Mutation-Filtering Approach

The goal of this approach was to further reduce the mutant count by identifying and eliminating redundant mutants. Mutation Testing is popular within the Software Testing community. A lot of research has been carried out by different researchers in improving the Mutation testing in terms of its computational cost. Different methodologies have been proposed to identify redundant mutants in conditional expressions for different types of mutation operators including ROR (Relational Operator Replacement), COI (Conditional Operator Inversion), SDL (Statement Deletion), etc. Among them, the research carried out by Kaminski on mutant reduction using logic-based testing [21] and also Chihiro & Shingo's [20] work on mutant reduction for conditional expressions within if-else and while statements,

was a major inspiration for our work in hardware mutation testing. Their work is based on a redundancy property of mutants, i.e., if a test suite that kills mutant $m0$ can also kill mutant $m1$, then we say that $m1$ is redundant. Following this observation, they focused on identifying such redundancy on ROR, COI, SDL mutation operators and proposed that COI and SDL mutation operators are redundant if a test suit kills ROR based mutants. Kaminski's work proposed that ROR mutation operator which generates seven mutants per relational operator (e.g., '>'), only three of the seven mutants are actually required, the rest become redundant. For example, let's say the original expression in a particular if-else statement is "z<0" and the corresponding mutations possible for this statement are listed in Table 4.1 [20]. For each mutant listed in this table, we have another column which gives the expression for the desired input that will differentiate the mutated expression from the original example. Now if we carefully observe the mutants 3,5, and 7, as suggested by Chihiro, the mutants 1,2,4, and 6 are redundant, that is, if we are able to achieve the test inputs that could detect mutants 3, 5, and 7, then the same test suite can detect the remaining mutants.

A similar approach was used in our work. RTL designs with heavy usage of conditional statements (if-else) benefit from such an approach. For all the relational expressions within if-else in the verilated RTL, we only generate 3 mutants instead of all 7. Hence per expression, we save approx. 4 mutants. We conducted experiments to verify this approach on ITC 99 circuits and we observe a marked improvement in some of the circuits like b15 and b14 in terms of mutant count and simulation time.

4.3 Experiments & Results

The proposed framework is developed in C++. All the experiments were conducted on a Linux (Kubuntu 4.13.3) machine with an Intel Core i7-6700K CPU @ 4GHz with 62.8 GB of

Sno	Original	Mutant	Input to Differentiate
1	$z < 0$	$z > 0$	$z \neq 0$
2	$z < 0$	$z \geq 0$	T
3	$z < 0$	$z \leq 0$	$z == 0$
4	$z < 0$	$z == 0$	$z <= 0$
5	$z < 0$	$z \neq 0$	$z > 0$
6	$z < 0$	<i>true</i>	$z \geq 0$
7	$z < 0$	<i>false</i>	$z < 0$

Table 4.1: ROR Mutants and Input to Differentiate for Original: $z < 0$

Benchmark	#Lines	#Branches	#PIs	#POs	#FFs
b06	128	24	2	6	9
b07	92	19	1	8	49
b10	167	32	11	6	17
b11	118	32	7	6	31
b14	509	211	32	54	245
b15	671	149	36	70	449

Table 4.2: Benchmark Characteristics

memory. We conducted experiments across benchmarks from ITC99 [6] detailed in Table 4.2. We chose to run our tool on those benchmarks of ITC99 which did not already reach 100% branch coverage by BEACON. The performance of the different methodologies presented here is compared with respect to the number of mutants, branch coverage, test generation time and test vector size.

The mutants generated in the preprocessing phase belong to the category of arithmetic, logical, relational, constants, interchanged signal/variable names for our work. The algorithm for the first pass of our test generation has the following parameters set: the number of ants is set to $K = 120$ to enable greater communication between the ants. The maximum rounds (repetitions) for the Global Search is set to $R = 10$. The number of cycles (length of vectors) used for each round (Max rounds here is 3) of the Local search is dependent on the size of the circuit. The pheromone evaporation rate is set to 90%, the initial pheromone deposit is

Table 4.3: Comparison of Different Approaches of Mutation Filtering

Bench	Number of Mutants			Branch Coverage %		Vector Size				Run Time (seconds)			
	Set 0	Set 1	Set 2	BEACON	Set 0	BEACON	Set 0	Set 1	Set 2	BEACON	Set 0	Set 1	Set 2
b06	151	93	55	95.83	95.83	1731	773	253	143	0.0054	0.262	0.0068	0.0045
b07	283	127	88	90	90	759	339	310	258	0.37	0.55	0.13	0.09
b10	720	323	273	93.75	96.88	3547	815	376	327	11.40	5.72	3.92	3.12
b11	693	395	310	96.88	96.88	1235	533	189	157	11.95	7.73	4.83	4.12
b14	2852	615	488	91.94	94.7	4381	1296	928	652	204.65	163.85	78.32	63.22
b15	3785	573	412	89.93	92.19	12917	7208	4813	2895	255.9	78.22	36.22	25.77

BEACON: No mutants used

Set 0: No Mutant Filtering

Set 1: First Filtering Approach

Set 2: Second Filtering Approach

Branch coverage for Set 0, Set 1, and Set 3 are identical

set to 1000 and maximum pheromone released by an ant is $P = 60$. The settings for the Second Pass Test Generation stage is similar to the one used in BEACON experiments. We next describe the experiments:

We performed three sets of experiments and compared the results against the original BEACON [24]. The first set of experiments is performed without any filtered mutants and demonstrate the advantage of the dual metric framework over BEACON. In addition, it aims to justify the motivation of developing the two filtering approaches. The second set of experiments extends the first experiment by including the first approach of Mutation Filtering (Set 1) and we evaluate the total number of mutants generated, the test vector size and the overall runtime of the tool to generate test suites while maintaining the same improved branch coverage for each benchmark. Finally, the third set of experiments apply the second approach of Mutation Filtering (Set 2).

Table 4.3 presents all the results of all the experiments. For each circuit, the number of mutants used, branch coverage, test vector size, number of mutants used, and the total run times are reported for the original BEACON, Set 0, Set 1, and Set2, respectively. We note that the branch coverages for Set 0, 1, and 2 are all the same, so only one column (Set 0) is included. Note that we were able to achieve equal or higher coverages compared to

BEACON in all instances. In addition to coverage, we report our observations with respect to the following performance features in detail:

Number of Mutants

We observe that the number of mutants by Set 1 and Set 2 was greatly reduced relative to Set 0. In b14 and b15, more than 80% of the mutants in Set 0 could be removed without loss in the coverage! This filtering of mutants contributes to reductions in run-times, test lengths while maintaining (or improving) coverage.

Test Generation Time

In terms of execution time, for certain circuits like b06 and b07, the test generation time for Set 0 increased because BEACON already achieved very good results. The execution times were notably shorter for the other circuits. For Set 1, the runtime is reduced by another 40% for benchmarks b06, b07, b11, b14 and b15 when compared to the *Set 0* results. The marginal improvement in b10 is because the second stage of the dual metric tool incurred more time as the mutants in the control flow path were not effective in helping to reach the required branch coverage. The runtime for Set 2 is further decreased as the total number of mutants is reduced. We see that the overall reduction in runtime of the tool is decreased up-to 85%. For b10, we see that the improvement from the first approach is negligible. The reason is that the second approach is beneficial for those designs which have many conditional statements and b10 comparatively has less in number.

Test Length

We observe that irrespective of either the branch coverage improved or maintained the same, the test vector size decreased significantly for all the ITC99 circuits in Set 0, Set 1 and Set 2. For the Set 0 cases, the test vector lengths were reduced by more than 50% for all but one circuit, b15. Set 1 and Set 2 further reduce the test lengths. We came across an important finding that for circuit b10, even though the runtime did not show a pronounced improvement, only less than 10% of the BEACON vectors were needed to achieve this final higher coverage!.

Chapter 5

PCB Simulation

During the course of my thesis, I had an opportunity to apply a part of my research at Tesla Inc., Palo Alto, California. At Tesla, I pursued an internship for 4 months with the Autopilot Hardware team, working on developing a methodology to simulate a PCB (Printed Circuit Board). In this project, I had an opportunity to conduct some experiments of my dual metric test generation methodology, on industry application circuits to understand the limitation and scope of my research work. Further, in this chapter, I will describe the company, project work in detail and its contribution to my research.

5.1 Motivation

”Tesla, Inc. (formerly Tesla Motors) is an American multinational corporation that specializes in electric vehicles, energy storage and solar panel manufacturing based in Palo Alto, California. Founded in 2003, the company specializes in electric cars, lithium-ion battery energy storage, and residential photovoltaic panels (through the subsidiary company SolarCity). The additional products Tesla sells include the Tesla Powerwall and Powerpack batteries, solar panels, and solar roof tiles.” [33]

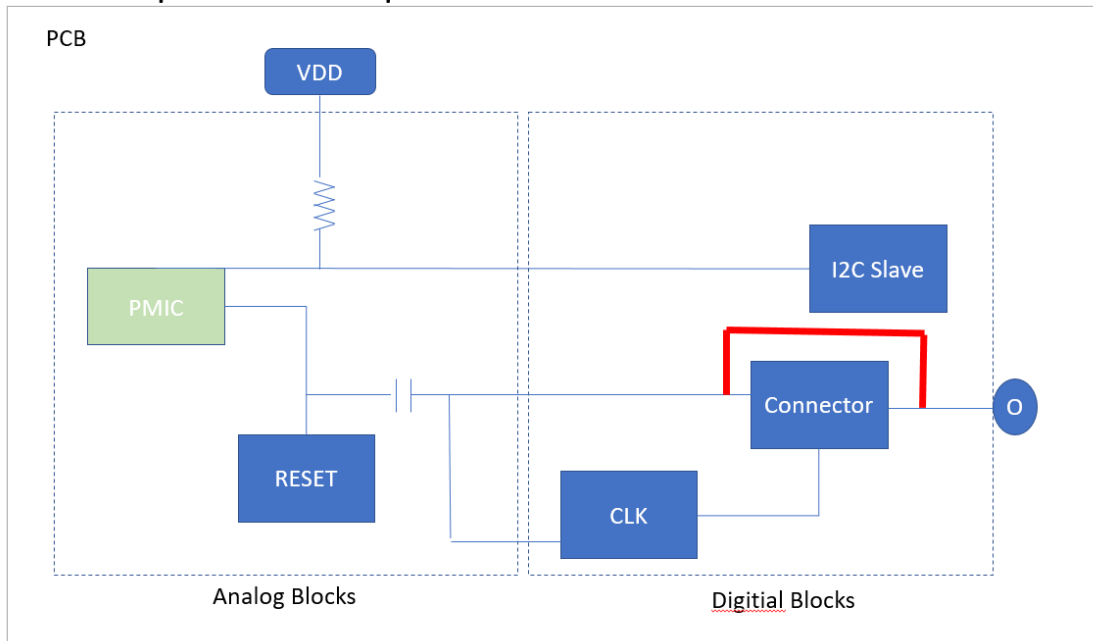


Figure 5.1: Simple PCB

Modern vehicles, including Tesla's category defining platforms, have complex systems that allow the firmware to control virtually every part of the vehicle. To enable such features in systems, the hardware involved also tends to be very complex in design. The design complexities are due to the presence of several components which requires constant interaction with each other. Hence, validation of the such systems is very crucial to ensure robust operation of the various systems.

A simple PCB (Printed Circuit Board) contains many analog components and digital components as shown in Figure 5.1. In this figure, we see a red marked wire connecting the two ends of a connector unit. In the expected PCB, the pins must be shorted. For example, in a potentially faulty design, the pins may remain unconnected due to a mistake while drawing the schematic. When the circuit board is taped out from the fabrication unit which takes roughly up to 2-3 weeks, Bring-up phase starts with screening the actual board by multiple tests to carry out initial functional checks, connectivity checks. During this phase especially

with respect to the connectivity check, if a single connection goes missing or not properly connected and not get caught at this point of validation cycle i.e. for our current illustrative system, if the pins are not connected then the output could depend on the details within the connector unit and, it could lead to unexpected results. Even fixing the bug, would involve repeating the process of redoing the schematic out, wait for 2-3 weeks and then again verify on Bring-up which means that valuable time is lost. A verification methodology where such supposed to be minor issues, could get caught faster and even before bring-up phase and save the unnecessary waiting time will be very valuable to integrate into the design process.

To be able to provide a canonical format for the overall verification flow, we chose to create a Verilog model from the schematics which were used to create the circuit board. Prior to sending the schematic to fabrication, using this process, we are able to verify the basic connectivity and functional check on the Verilog model of the board and notify any changes required to before sending the PCB for fabrication. In the next section, we will explain the methodology in detail.

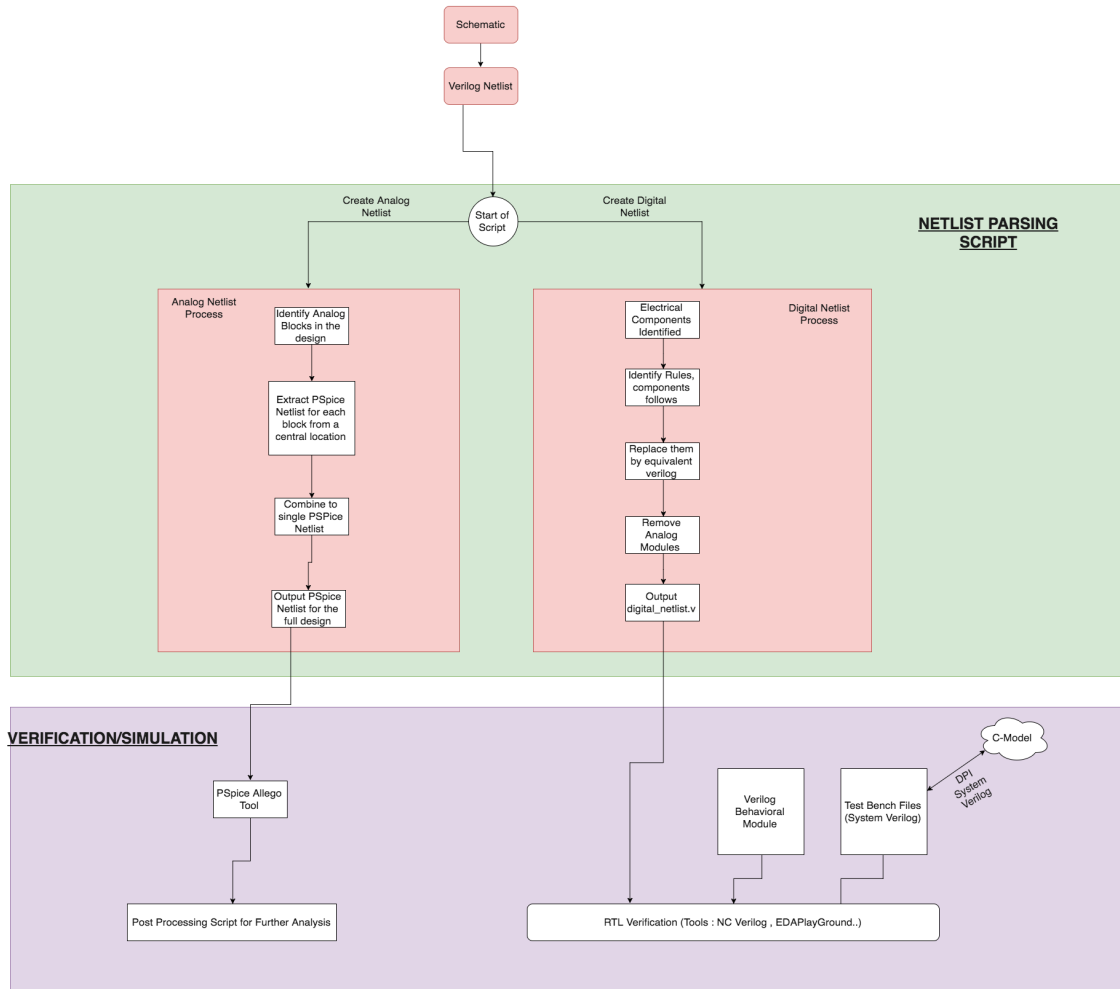


Figure 5.2: Methodology

5.2 Methodology

The schematic is the starting point of our methodology as shown in Figure 5.2. It is first converted to a Verilog Netlist using Cadence Virtuoso Schematic [?]. Then a post-processing script (Figure 5.3) is written to identify and separate the analog and digital components.

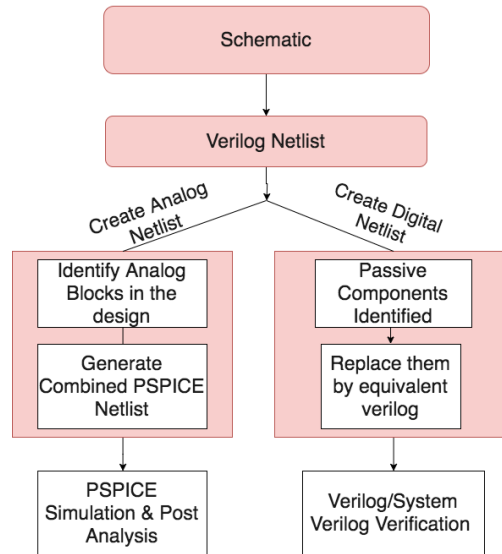


Figure 5.3: Post Processing Flow

Finally, we get a digital netlist file where, all the passive components (resistor, capacitor, inductor) are converted into their digital equivalent models. For example, if there is a resistor connecting two ICs, then it can be approximated to a short circuit in the digital world, i.e. resistor is considered just a wire. Similar to these, there are many different arrangements for various components which are modified. All the individual ICs, like I2C modules, HDMI modules etc. are replaced by their Verilog functional models according to the given specifications.

We end up with a fully loaded Verilog model containing all the digital components of the PCBA being verified. These models are complex in terms of the number of case statements, with multiple clock signals and I/O signals, since they represent the actual PCBA used in the Infotainment systems. This circuit hence could be a very good example to experiment with our verification methodology developed in my research. These models were passed through the Dual Metric Test Generation framework as explained in previous sections and test vectors were generated which aimed at achieving high branch coverage. To also enable further experiments on verifying the functional check, we created a System Verilog Test bench,

Bench	Number of Branches	Branch Coverage %		Execution time(s)	
		BEACON	Ours	BEACON	Ours
PCB_0	185	93.5	95.6	0.015	0.045
PCB_1	247	88.2	88.2	3.86	2.88
PCB_2	573	95.9	95.9	166.43	143.22
PCB_3	821	92.5	92.5	377.11	379.28

Table 5.1: Branch Coverage for PCB Simulations

also with an ability to interface with the external environment via Direct Programming Interface(DPI).

5.3 Results

We conducted experiments on multiple types of PCBs, having a combination of multiple units like I2c, HDMI, PCI-E. Verilog models were generated for these blocks and passed through the Dual Metric Test Generation Framework to generate test vectors aiming at branch coverage as the metric. Results are tabulated in 5.1. We observed that using our approach, we were able to generate test vectors and achieve coverage up to 96.7%. We also presented the results of the branch coverage achieved by the original BEACON and observe that the new methodology performs better than that by BEACON with respect to Branch Coverage and execution time. The speedup using Dual Metric Framework is up-to 2x of the same achieved by BEACON.

Chapter 6

Conclusion

In this chapter we are giving a concluding summary of the work presented in this thesis. We are also providing a discussion on future directions of this work.

6.1 Limitations and Future Work

In our work, we only considered a small subset of mutant space from the superset of infinite mutant space. Mutant categories included arithmetic, logical, relational and constants. This small subset definitely limited the search space in reaching hard-to-reach branches. Having a much broader set of mutants can help achieve the same branch coverage in a lesser amount of time and can also provide a reduction in the vector length. This is one of our future work where we will be also implementing other mutant types like statement deletion based mutants. There is another advantage of having a broader set of mutants. The first stage is aimed at generating test vectors based on Mutation Coverage. Hence, enabling a broader set of mutants can provide the user with another application of this dual metric framework. The user can not only generate test vectors with branch coverage as the metric but also can consider generating test vectors only with mutation coverage as the metric.

Secondly, we have a limitation from the Verilator's side in terms of converting the large-width signals. In RTL designs, signals which have the bit width of more than 64 can lead to an incorrect conversion due to this limitation. So this is one of the items listed as well in our

future work.

Also, currently we analyze one mutant at a time by creating a mutated design per mutant in the preprocessing step. Then in the first stage, one pair of mutated and original design are considered and are evaluated to generate test vectors aiming at the mutant detection. Hence, if we have 1000 mutants, then this process is repeated for 1000 times, resulting in higher execution time. So our future work involves building up techniques to optimize this process by enabling parallelism in the simulation, where multiple such pairs of designs can be analyzed. Also, another direction which can be thought of is analyzing multiple mutants in one single file at one go.

6.2 Conclusion

In this thesis, we proposed a dual metric test generation methodology for the Register Transfer Level(RTL) and also techniques to optimize the performance of dual metric test methodology with respect to runtime and test vector size. Dual Metric test methodology uses two coverage metrics: branch coverage and mutation coverage to enhance the branch coverage of the given RTL and generate test vectors to achieve the same. In previous techniques like BEACON, though they helped achieve a high branch coverage, still there were some hard-to-reach branches uncovered. This was the motivation behind the proposed technique. Mutations by nature in a design have the capability to modify the data and control flow path of a test simulation. Using the help of this important property of mutants and proper path guidance algorithms like Genetic Algorithms and Ant Colony Optimization, we are able to cover those uncovered hard-to-reach branches. Experiments, when conducted with ITC99 benchmarks, showed an improvement of up-to 70% in branch coverage when compared to the previous techniques.

But for some circuits, this added benefit came with a cost in execution time. Following which, in the subsequent chapters we proposed two filtering techniques to optimize the first stage of the dual metric framework. These techniques help further reduce the time taken by the test generation framework, when tests are generated without any mutant filtering, by eliminating redundant and unnecessary mutants (up to 80% mutant reduction). We also observed that the structure of the RTL design plays an important role in evaluating the advantage of these techniques. In many ITC99 circuits, an improvement of up-to 85% is seen in the test generation runtime over the existing method. Also, the size of final test vectors showed a reduction up to 78% with no loss in branch coverage.

Bibliography

- [1] *The chip that changed the world.* URL <https://www.nytimes.com/2008/09/19/opinion/19iht-eddas.1.16308269.html>.
- [2] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM. doi: 10.1145/800028.808479. URL <http://doi.acm.org/10.1145/800028.808479>.
- [3] R. Arora and M. S. Hsiao. Enhancing sat-based bounded model checking using sequential logic implications. In *17th International Conference on VLSI Design. Proceedings.*, pages 784–787, 2004. doi: 10.1109/ICVD.2004.1261028.
- [4] Boris Beizer. *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990. ISBN 0-442-20672-0.
- [5] V. Bertacco. Distance-guided hybrid verification with guido. In *2006 IEEE International High Level Design Validation and Test Workshop*, pages 151–151, Nov 2006. doi: 10.1109/HLDVT.2006.319979.
- [6] F. Corno, M. S. Reorda, and G. Squillero. Rt-level itc'99 benchmarks and first atpg results. *IEEE Design Test of Computers*, 17(3):44–53, Jul 2000. ISSN 0740-7475. doi: 10.1109/54.867894.
- [7] R. A. DeMilli and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, Sep 1991. ISSN 0098-5589. doi: 10.1109/32.92910.

- [8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978. ISSN 0018-9162. doi: 10.1109/C-M.1978.218136.
- [9] S. Devadas, A. Ghosh, and K. Keutzer. An observability-based code coverage metric for functional simulation. In *Proceedings of International Conference on Computer Aided Design*, pages 418–425, Nov 1996. doi: 10.1109/ICCAD.1996.569832.
- [10] M. Dorigo, V. Maniezzo, and A. Coloni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, Feb 1996. ISSN 1083-4419. doi: 10.1109/3477.484436.
- [11] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, Nov 2006. ISSN 1556-603X. doi: 10.1109/MCI.2006.329691.
- [12] H. Foster. *The 2010 Wilson Research Group Functional Verification Study*, 2011.
- [13] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, July 1977. ISSN 0098-5589. doi: 10.1109/TSE.1977.231145.
- [14] G. Al Hayek and C. Robach. From specification validation to hardware testing: a unified method. In *Proceedings International Test Conference 1996. Test and Design Validity*, pages 885–893, Oct 1996. doi: 10.1109/TEST.1996.557150.
- [15] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0262082136.

- [16] M. S. Hsiao, E. M. Rudnick, and J. H. Patel. Automatic test generation using genetically-engineered distinguishing sequences. In *Proceedings of 14th VLSI Test Symposium*, pages 216–223, Apr 1996. doi: 10.1109/VTEST.1996.510860.
- [17] M. S. Hsiao, E. M. Rudnick, and J. H. Patel. Sequential circuit test generation using dynamic state traversal. In *Proceedings European Design and Test Conference. ED TC 97*, pages 22–28, Mar 1997. doi: 10.1109/EDTC.1997.582325.
- [18] M. S. Hsiao, E. M. Rudnick, and J. H. Patel. Fast static compaction algorithms for sequential circuit test vectors. *IEEE Transactions on Computers*, 48(3):311–322, Mar 1999. ISSN 0018-9340. doi: 10.1109/12.754997.
- [19] Michael S. Hsiao, Elizabeth M. Rudnick, and Janak H. Patel. Dynamic state traversal for sequential circuit test generation. *ACM Trans. Des. Autom. Electron. Syst.*, 5(3): 548–565, July 2000. ISSN 1084-4309. doi: 10.1145/348019.348288. URL <http://doi.acm.org/10.1145/348019.348288>.
- [20] C. Iida and S. Takada. Reducing mutants with mutant killable precondition. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 128–133, March 2017. doi: 10.1109/ICSTW.2017.29.
- [21] Gary Kaminski, Paul Ammann, and Jeff Offutt. Improving logic-based testing. *J. Syst. Softw.*, 86(8):2002–2012, August 2013. ISSN 0164-1212. doi: 10.1016/j.jss.2012.08.024. URL <http://dx.doi.org/10.1016/j.jss.2012.08.024>.
- [22] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8): 870–879, August 1990. ISSN 0098-5589. doi: 10.1109/32.57624. URL <http://dx.doi.org/10.1109/32.57624>.
- [23] D. Krishnaswamy, M. S. Hsiao, V. Saxena, E. M. Rudnick, J. H. Patel, and P. Banerjee.

- Parallel genetic algorithms for simulation-based sequential circuit test generation. In *Proceedings Tenth International Conference on VLSI Design*, pages 475–481, Jan 1997. doi: 10.1109/ICVD.1997.568180.
- [24] M. Li, K. Gent, and M. S. Hsiao. Design validation of rtl circuits using evolutionary swarm intelligence. In *2012 IEEE International Test Conference*, pages 1–8, Nov 2012. doi: 10.1109/TEST.2012.6401556.
- [25] P. Lisherness and K. T. Cheng. Scemit: A systemc error and mutation injection tool. In *Design Automation Conference*, pages 228–233, June 2010. doi: 10.1145/1837274.1837333.
- [26] L. Liu and S. Vasudevan. Efficient validation input generation in rtl by hybridized source code analysis. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011. doi: 10.1109/DATE.2011.5763253.
- [27] G. Moore. *Progress in digital integrated electronics*, 1975. URL [ElectronDevicesMeeting,1975International](#).
- [28] M. S. Nsiao, E. M. Rudnick, and J. H. Patel. Fast algorithms for static compaction of sequential circuit test vectors. In *Proceedings. 15th IEEE VLSI Test Symposium (Cat. No.97TB100125)*, pages 188–195, Apr 1997. doi: 10.1109/VTEST.1997.600260.
- [29] S. Pinto and M. S. Hsiao. Rtl functional test generation using factored concolic execution. In *2017 IEEE International Test Conference (ITC)*, pages 1–10, Oct 2017. doi: 10.1109/TEST.2017.8242038.
- [30] A. Sen and M. S. Abadir. Coverage metrics for verification of concurrent systemc designs using mutation testing. In *2010 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 75–81, June 2010. doi: 10.1109/HLDVT.2010.5496659.

- [31] Y. Serrestou and V. B. C. Robach. Functional verification of rtl designs driven by mutation testing metrics. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 222–227, Aug 2007. doi: 10.1109/DSD.2007.4341472.
- [32] Wilson Synder. *Verilator*. URL <https://www.veripool.org/wiki/verilator>.
- [33] Wikipedia contributors. Tesla, inc. — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Tesla,_Inc.&oldid=844414263, 2018. [Online; accessed 5-June-2018].
- [34] Tao Xie, Wolfgang Mueller, and Florian Letombe. Efficient mutation-analysis coverage for constrained random verification. In Mike Hinchey, Bernd Kleinjohann, Lisa Kleinjohann, Peter A. Lindsay, Franz J. Rammig, Jon Timmis, and Marilyn Wolf, editors, *Distributed, Parallel and Biologically Inspired Systems*, pages 114–124, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15234-4.
- [35] Liang Zhang, M. R. Prasad, and M. S. Hsiao. Incremental deductive inductive reasoning for sat-based bounded model checking. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pages 502–509, Nov 2004. doi: 10.1109/ICCAD.2004.1382630.
- [36] Y. Zhou, T. Wang, T. Lv, H. Li, and X. Li. Path constraint solving based test generation for hard-to-reach states. In *2013 22nd Asian Test Symposium*, pages 239–244, Nov 2013. doi: 10.1109/ATS.2013.52.