Branch-guided Metrics for Functional and Gate-level Testing

Vineeth V. Acharya

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
Chao Wang
A. Lynn Abbott

20 February, 2015
Blacksburg, Virginia

Keywords: Branch Coverage, Algorithms, Fault Coverage,
State-augmented Branch Score, Functional Test Generation

Copyright 2015, Vineeth V. Acharya

Branch-guided Metrics for Functional and Gate-level Testing

Vineeth V. Acharya

(ABSTRACT)

With the increasing complexity of modern day processors and system-on-a-chip (SOCs), designers invest a lot of time and resources into testing and validating these designs. To reduce the time-to-market and cost, the techniques used to validate these designs have to constantly improve. Since most of the design activity has moved to the register transfer level (RTL), test methodologies at the RTL have been gaining momentum. We present a novel functional test generation framework for functional test generation at RTL. A popular software-based metric for measuring the effectiveness of an RTL test suite is branch coverage. But exercising hard-to-reach branches is still a challenge and requires good understanding of the design semantics. The proposed framework uses static analysis to extract certain semantics of the circuit and uses several data structures to model these semantics. Using these data structures, we assist the branch-guided search to exercise these hard-to-reach branches. Since the correlation between high branch coverage and detecting defects and bugs is not clear, we present a new metric at the RTL which augments the RTL branch coverage with state values. Vectors which have higher scores on the new metric achieve higher branch and state coverages, and therefore can be applied at different levels of abstraction such as post-silicon validation. Experimental results show that use of the new metric in our test generation framework can achieve a high level of branch and fault coverage for several benchmark circuits, while reducing the length of the vector sequence.

This work was supported in part by the NSF grant 1016675.

$\sim$ *To my Family* $\sim$

Acknowledgments

I would like to thank my research advisor, Dr. Michael S. Hsiao, for mentoring and guiding me during my research. It was his exceptional teaching in the course "Testing of Digital Systems" in Fall 2012 that inspired me to pursue my masters' thesis under his tutelage. I am deeply inspired by his extensive knowledge, artful thinking, dedication and amiable nature. I am honored to have gotten a chance to work with him.

I would like to thank Dr. Chao Wang and Dr. A. Lynn Abbott for serving on my thesis committee.

I would like to thank the denizens of 24-I, Vivek Jayabalan, Michael Dylan D'souza, Shashank Kidiyur Sathish, Anupriya Gupta, Sudarshan Aji, Tejaswi Gode and Chaitra Raghunath for being wonderful roommates and making Blacksburg a home away from home. My sincere gratitude to my best friends, Anupriya and Vivek for all your encouragement and support through good times and tough times.

Special thanks to Sarvesh Prabhu, Michael Dylan D'souza, Sharad Bagri for all the technical and philosophical discussion inside and outside the lab. I would also like to thank Arijit Chattopadhyay, Krishna Chaitanya Pabbuleti and Deepak Mane for all the fun times.

I would like to express my sincere gratitude to my parents Vadiraj V. Acharya and P. Jayanthi, for believing in me every step of the way, and my dear sister Vibha, for being there

for me throughout my life. You guys are the best. I would also like to that my extended family for their love, support and encouragement.

Finally, I would also like to thank my PROACTIVE labmates, Kelson Gent, Sarmad Tanwir, Indira Priyadarshini, Avinash Desai and Shuchi Pandit for creating and maintaining a fun and intellectual environment in the lab.

Vineeth V. Acharya

February 2015

Contents

1 Introduction **1**

1.1 Contributions . . . 3

1.2 Thesis Organization . . . 4

2 Background **6**

2.1 Design Verification . . . 6

2.1.1 Functional Verification at the RTL . . . 7

2.2 Prior Work in Test Generation . . . 8

2.2.1 Test Generation at the RTL . . . 8

2.2.2 Test Generation at the gate level . . . 10

2.3 Genetic Algorithms . . . 13

2.4 Verilator . . . 15

3 Branch-guided Functional Test Generation for RTL **17**

3.1 Introduction . . . 17

3.2 Related Work . . . 18

3.3 Test Generation Framework . . . 20

3.3.1 Branch Hierarchy Tree . . . 20

3.3.2 Assignment Table . . . 23

3.3.3 Branch Transition Graph . . . 25

3.3.4 Abstract state . . . 27

3.4 Genetic Algorithm Search . . . 28

3.4.1 GA: Stage 1 28
3.4.2 GA: Stage 2 29
3.4.3 Fitness functions 31
3.5 Results 32

4 A New Metric for Functional Test Generation **35**
4.1 Introduction 35
4.2 Motivation 37
4.3 State-augmented Branch Score 39
4.4 Test Generation 41
4.4.1 GA - Stage 1 42
4.4.2 GA - Stage 2 43
4.4.3 Fitness functions 44
4.5 Results 46
4.5.1 Algorithmic Settings 47
4.5.2 Vector Generation 47

5 Conclusion **50**
5.1 Conclusion 50
5.2 Future Work 51

Bibliography **52**

List of Figures

2.1 Example Verilog code and Verilator C++ code 16

3.1 Example Verilog code and Verilator C++ output 22
3.2 Branch Hierarchy Tree 23
3.3 Assignment Table 24
3.4 Finite State Machine and Branch Transition Graph 25

4.1 Example Verilog RTL code snippet 38

List of Tables

3.1 Benchmark Characteristics 33
3.2 Branch Coverage and Comparison with Prior Work 33

4.1 Comparison of the three metrics 39
4.2 Benchmark Characteristics 47
4.3 Comparison of branch coverage vs SABS 48
4.4 Comparison of SABS with Prior Work 49

List of Algorithms

4.1 State-augmented branch score computation 40

Chapter 1

Introduction

Advances in VLSI and manufacturing process technology have resulted in the constant shrinking of transistors. Following Moore's Law [1], the number of transistors we can afford to put on a chip continues to increase, thereby making very complex designs feasible. In today's information age, technology is ubiquitous and has been interweaved into our everyday lives. With the advent of the Internet-of-Things, these chips find themselves in a wide range of applications, ranging from mission-critical systems such as automobiles and health-care to personal computing such as wearables and mobile devices to large-scale systems such as smart grids. Thus, ensuring that they are void of errors, bugs and defects is a must. The increase in design complexity of these modern-day processors and system-on-a-chip (SOC) has resulted in higher design validation and verification costs. It is estimated that design verification consumes up to 50% of the design efforts and is expected to continue rising [2]. Simulation is the primary technique used in today's verification practices, and therefore, effective test patterns are needed so that all parts of the circuit can be exercise.

In order to meet the time to market for these complex chips, techniques used for design verification have to constantly improve. Over the last couple of decades, the maturity of

logic synthesis, shorter code length and increased readability of register-transfer level (RTL) code has enabled the paradigm shift of moving the design activity from the gate level to the RT-level. A side benefit of this shift is that validation can also be conducted at this level, in which the design intent is better modeled when compared to the gate-level counterpart. Successful validation of the design requires an effective suite of test inputs (or stimuli). For example, random stimuli can be used to cover a portion of the design space, but many hard to reach areas and corner cases remain unexercised. In order to reach these corner cases, random stimuli is often augmented with manual directed testing, in which tests are written manually for each corner case. However, manual testing consumes a significant amount of time, further adding to the cost of validation. It is also error-prone and requires expert knowledge of the design to produce effective vectors. With the increasing complexity of the design, there is a need for automated techniques to generate intelligent functional vectors.

Because corner cases may not be easy to describe or quantify, metrics that try to correlate the effectiveness of a test suite and the extent to which the design has been exercised have been proposed. In this regard, software-based code coverage metrics, such as branch coverage, state coverage, path coverage, statement coverage, etc. have been used. Unlike software, the RTL description of the hardware succinctly describes one cycle of its operation. Therefore, branch coverage has been a popular and practical metric used because maximum branch coverage implies that all valid control states have been reached in the RTL design.

In the recent years, several automated test generation methods for functional verification have been proposed. These methods can be broadly divided into two classes, deterministic methods and simulation-based methods. Deterministic methods generally use formal verification techniques such as bounded model checking (BMC) and symbolic execution to generated function tests. Simulation-based methods on the contrary employ simulation and heuristics to choose from a set of several candidate solutions. Heuristics such as Genetic

Algorithms (GAs) and Ant Colony Optimization (ACO) have been successfully used in the past.

Although software based metrics such as branch coverage have been widely used in design validation, the degree of correlation with catching defects or bugs is not clear. For example, is 100% branch coverage sufficient? If not, what other practical metrics can be used or combined with branch coverage? One could reason that adding gate-level coverage (such as stuck-at) could enhance the metric. However, simulation at the gate-level is significantly more expensive than at the RTL, and hence something different would be preferable. This motivated us to formulate a metric that can achieve high coverages at both the RTL (branches) and gate-level (stuck-at faults) without resorting to gate-level simulation. Using this new metric, our test generator attempts to generate stimuli which are more effective at catching defects.

1.1 Contributions

In this thesis, we present a novel functional test generation method at the RTL which incorporates intelligence from the design to reach hard corner branches. The RTL code is first cross-compiled into C++ and instrumented for branch coverage using Verilator, an open source tool. Simulation in C++ is significantly faster than Verilog, thus helps reduce the overall the computation time. It uses a branch-guided GA-based search that comprises of a branch-independent stage, which aims for maximal branch coverage, and a targeted branch-dependent search for the uncovered branches. In order to assist the GA on reaching hard branches, we present a new framework which uses static analysis to extracts circuit semantics, such as flow of control and the hierarchy between the branches. The framework also provides a few data structures which efficiently model these semantics of the RTL in

terms of branches. These data structures are then employed during the search to distinguish between the various candidate solutions. The proposed method has proven to provide input sequences which are shorter by up to two orders of magnitude in length, while maintaining similar branch coverages on a number of circuits.

In the second part of this thesis, we extend the functional test generation method so that the generated test stimuli can also be applied at various levels of abstraction, including as function verification and non-scan post-silicon validation. To do so, we present a new metric which uses state-level Since RTL allows for arithmetic and relational operations in addition to logical operations, multiple state values might correspond to the same branch. The proposed metric uses this state-level information in addition to the branches, and hence called *state-augmented branch score*. Unlike branch and fault coverage, the new metric is expressed as an absolute score, instead of a percentage value. Test sets that achieve a high score on this metric will achieve a high branch coverage as well as state coverage. The proposed metric has been found to be effective in expressing coverage at the RTL (branches) and gate-level (stuck-at faults). The test generation method based on this metric is able to successfully generate vectors similar coverages to previous methods on a number of circuits, while achieving shorter test sequence lengths.

1.2 Thesis Organization

The rest of this is organized as follows.

Chapter 2 discusses the relevant past work and provides the background material for Genetic Algorithms and Verilator.

Chapter 3 provides a detailed description of the test generation framework including the

circuit semantics extracted, the data structures proposed and the GA-based search. The results for branch coverage are also reported in this chapter. This work will also appear in ETS 2015 [3].

Chapter 4 presents the motivation and the algorithm for the new proposed metric, state-augmented branch score. It also includes a comparative study on the different prior techniques with different metrics. Finally, the results for the new metric is reported in this chapter. The work done in this chapter has been submitted to the ITC 2015.

V. V. Acharya and M. S. Hsiao, "A New Metric and Framework for Functional Test Generation,", under review, *IEEE International Test Conference (ITC) 2015.*

Chapter 5 concludes this thesis and provides recommendations for future work.

Chapter 2

Background

2.1 Design Verification

The intent of design verification is to verify that the design conforms to the specifications. The design verification process consumes up to 70% of the total product development time [4]. The three most common approaches to design verification are:

1. **Logic simulation/emulation** and circuit simulation, in which the functionality and timing of the design is checked by simulation or emulation;
2. **Functional verification**, in which functional models are developed to describe the functionality of the design, which are then checked against the behavioral specification without considering timing accuracy; and
3. **Formal verification**, in which the functionality and design properties are checked against a *golden model*.

While logic simulation verifies timing accuracy, functional verification only verifies the cycle accuracy. Therefore, logic simulation is the more dominant metric used. However, functional verification of register-transfer level (RTL) descriptions has the following advantages.

2.1.1 Functional Verification at the RTL

The register transfer level (RTL) is a level of abstraction which models a circuit in terms of the flow of signals (or data) between registers in the circuit. To describe a circuit at the RTL, a hardware description language (HDL) such as Verilog or VHDL is used, which is also commonly called RTL code. The RTL code succinctly describes one cycle of execution, and includes relational and arithmetic operations in addition to logical operations. This makes RTL code shorter in length and more readable.

Higher level descriptions (such as RTL) of the circuit have fewer implementation details but more explicit functional information than lower level descriptions (such as gate-level). Therefore, functional models can be developed more easily from RTL than gate-level descriptions. The simulation of these functional models is much faster than the logic simulation of its gate-level counterparts, thereby compensating the loss of accuracy with reduced simulation times.

Additionally, the reduction in design verification time implies that a more thorough verification can be performed, thereby improving the quality of the design. Moreover, these models are smaller and have more functional information than gate-level netlists, making it is easier to detect, locate and correct any errors. Since RTL descriptions are available early in the design phase, functional verification at the RTL results in catching bugs early in the design phase. As a side benefit, the test stimuli developed for design verification of the RTL

and the corresponding output responses obtained are used for functional testing during the manufacturing.

2.2 Prior Work in Test Generation

In this section, we present the prior work in the field of test generation for sequential circuits. Although there has been a lot of research done in this field, automatic test pattern generation (ATPG) for sequential circuits is still an extremely hard problem. To alleviate this problem, design for testability (DFT) techniques such as scan-based testing are widely used in practice. In DFT, additional logic is added into the circuit that is active only in the test mode, and disabled in the functional mode. There are several trade-offs with using DFT techniques, primarily in terms of area and performance overhead, and significantly longer test application time. In this section, we will limit our discussion to non-scan based ATPG techniques which have been proposed in the past.

To tackle the problem of sequential ATPG, test generation methods have targeted various levels of circuit abstraction. The 2 most commonly targeted levels of abstraction, namely the register transfer level and the gate level, are discussed below.

2.2.1 Test Generation at the RTL

Several approaches have been proposed for test generation at the RTL. Corno et al. proposed a GA-based ATPG in [5], where the input RTL description is first instrumented and then simulated using a commercial simulator to produce an execution trace. The underlying GA-based procedure then interacts with the simulator via the trace files, and tries to generate an input sequence which executes a target statement or branch in the RTL code which has not

yet been reached. While trying to achieve maximum statement coverage, the algorithm is unable to distinguish between easy and hard branches. Therefore, many of the hard-to-reach corner cases remain unreached.

HYBRO is a coverage-driven technique, proposed in [6], which combines dynamic and static analysis to generate high coverage input vectors. The RTL code is instrumented and an execution trace is obtained by a concrete simulation. From the trace, the symbolic expression and the initial guard branches are extracted. The guards are then mutated and fed to a Satisfiability Modulo Theory (SMT) solver to obtain satisfiable input assignments. This approach allows HYBRO to effectively explore the design and achieve high branch coverage, but would require the control-flow graph to be unrolled several times before the target branch can be reached. Such symbolic execution may lead to path explosion, and limits the number of cycles that can be unrolled. Therefore, the branches which require longer input sequences are not reached.

On the other hand, BEACON[7] is a simulation-based design validation technique which uses a marriage of ant colony optimization (ACO) and evolution as a heuristic to achieve high branch coverage. The RTL code is instrumented with unique counters for each branch using Verilator[8] to help determine the branches which have been reached for a given input sequence. These input sequences resemble ants traversing various parts of the circuit and depositing pheromones. A high counter value for a particular branch is analogous to excess pheromones being deposited on the branch because it is reached by multiple ants. Such branches are considered easy to reach, and are trimmed from the search space so that the ants can target the harder branches. In order to leverage the merits of formal verification and simulation based techniques, several semi-formal techniques have been proposed. A hybrid extension to BEACON was proposed in [9] to make faster and better decisions. As a preprocessing step, the control-flow graph (CFG) of the RTL design is extracted. Then, a

frontier search is invoked periodically using an SMT-based bounded model checker (BMC) which helps navigate the narrow search paths that might be missed with random search.

In addition to branch coverage, there have been several abstraction guided simulation techniques for design validation which used semi-formal techniques [10–14]. Using an abstract model of the original circuit, some corner cases of the circuit can be exercised. But circuit abstraction results in some inherent loss of information. This leads to certain dead-end abstract states begin scored well, but actually resulting in a false path to the target corner state. These high scoring dead-end states cause the simulation to digress from the real paths to these necessary corner states.

2.2.2 Test Generation at the gate level

Once manufactured, every chip needs to be tested for manufacturing defects. These defects might occur due to several factors such as process variation, silicon defects, oxide defects etc. These physical defects in the chip are modeled as faults at the gate-level. To model the various types of defects, several fault models are used such as stuck-at faults, bridging faults, transition faults and path delay faults. The most commonly used fault model is the single stuck-at fault model where the defect has an effect of holding a signal in the circuit to a constant logic 0 or logic 1. It has been shown that a test set that is able to detect all single stuck-at faults has a high probability of detecting faults based on the other fault models as well [15].

Since defects in the silicon are modeled as faults at the gate level, and therefore most post-Si validation methods target the gate-level description of the circuit, called netlists. Sequential ATPG techniques are broadly classified into two categories, deterministic and simulation-based. Deterministic techniques systematically sweep through all the faults, usually targeting

a single fault at a time. For each target fault, it assigns values to the signals in the circuit such that the fault effect is propagated to a primary output. Since propagation of these fault effects to the PO might take multiple cycles, the circuit is sometimes unrolled for multiple cycles or timeframes. With advances in formal verification techniques such as satisfiability (SAT) and bounded model checking (BMC), SAT-based deterministic techniques have become popular. In these techniques, the circuit is converted into a conjunctive normal formula (CNF) and the fault effect is added as a clause. The CNF is solved using a SAT solver to obtain a satisfiable assignment at the inputs such that the fault effect is propagated to a primary output. HITEC was one of the earliest deterministic ATPGs [16].

Simulation-based techniques on the other hand generate efficient vector sequences by using logic simulation and a guided search. Several heuristics have been used to guide the gate-level search, such as Genetic Algorithms (GAs) [17–21], spectral analysis [22, 23] and state-partitioning [24]. Deterministic techniques are usually more expensive in terms of computation and have an upper limit on the number of times the circuit can be unrolled. Simulation based techniques, on the other hand are much faster but, are unable to reach harder states without the right guidance from the heuristics. Conventional test generation techniques may generate really long sequences, which would result in long test application times. To reduce the lengths of these sequences, several test compaction techniques have also been proposed in [25, 26].

Although, most design validation techniques target RTL descriptions, gate-level design validation techniques have been proposed in [11–13]. These techniques use abstraction guided simulation to reach hard-to-reach corner states and employ ACO, cultural algorithms and GA respectively to guide the simulation. Since these techniques target gate-level descriptions, they miss out on the high-level information such as the control-flow in the circuit.

The high-level information present in the RTL description can be used to navigate the circuit

more effectively. On the contrary, the gate-level descriptions provide a better idea of the necessary areas in the circuit which must be exercised in order to achieve high fault coverage. Since, both approaches have information vital to test generation, researchers have proposed techniques which target both the gate-level and the RTL. To facilitate this mixed-level test generation, several data structures have been employed in literature. Assignment Decision Diagrams (ADDs) were used in [27, 28], in which the RTL description is converted into a graph-like structure called Assignment Decision Diagrams (ADD). Each element in the graph is identified, and a precomputed set of test vectors are applied depending on the type of the element. In [29], the justification and propagation is based on a polynomial formal model of sequential circuits. There have also been several methods proposed which target both levels of abstraction. For example, in [30, 31], test generation is first performed at the RTL and then passed onto the gate-level ATPG as seed values to target the remaining structural details. The advantage of a mixed-level ATPG is that the structural details available at the gate-level are complemented by the high-level operations and control information available at the RTL.

In [32], a mixed-level test generation method is proposed which can be applied for pre-silicon and post-silicon validation. It extends the ACO-based search proposed in BEACON [7] to the gate level, thereby extending the functional test generation method to make it a sequential ATPG. In addition to branch coverage, fault coverage is also employed as a metric to extend the ACO-based search at the gate level. By introducing a gate-level co-simulator, the fault excitations at the gate-level is associated with branch activation patterns at the RTL. This provides a feedback which influences the ACO search in the next iteration. The resulting mixed-level ATPG is able to achieve maximum branch coverage at the RTL, and higher fault coverage than previous non-scan based ATPG for many hard-to-reach circuits. However, the need for gate-level fault simulation can be a burden to the process.

2.3 Genetic Algorithms

Genetic Algorithm (GA) is a bio-inspired algorithm presented by John Holland in [33], a search heuristic used to generate solutions to optimization/search problems. It is a subset of a class of evolutionary algorithms, and has been employed in several practical applications such as machine learning, job/process scheduling, test generation, etc. GA is very effective in solving problems where the search space is vast, and the solution need not be optimal.

In a typical GA, there is a population of candidate solutions (or individuals) to a given problem, which are iteratively evolved into better solutions by altering properties of these individuals. The initial population generally starts with randomly generated individuals. In the beginning of every iteration, each individual is associated with a *fitness* value, which measures the goodness of the individual corresponding to how it solves the problem at hand. The function used to compute this fitness value is called the fitness function. Then, the population is operated upon by 3 operators, viz selection, crossover and mutation to give rise to a new population. *Selection* involves a stochastic method of selecting individuals from the current population, based on their fitness values. In *crossover*, the two or more individuals are combined to create offsprings of the next generation. Each offspring has a combination of the properties (or traits) of each parent. Once crossover is completed, the *mutation* operator introduces small probabilities of random mutations in the new generation of offsprings. Mutation, in small quantities, helps in moving the population away from potential local optima, which traditional hill climbing optimization algorithm tend to get stuck in. To maintain the quality of individuals over generations, a few fittest individuals from each generation are passed onto the next generation without any modifications. This process is called *elitism*.

This marks the end of one iteration or generation, and this new generation is used as the starting point for the next iteration. The algorithm terminates when

1. A fixed number of iterations/generations are reached, or
2. The individual with the highest fitness reaches a threshold fitness value, or
3. The maximum (or average) fitness has reached a plateau and future generations are unlikely to produce better results.

GA is a very useful technique which can be employed for solving problems where the search space is vast, and the solution need not be optimal. GA has been used previously in test generation, since the test vectors generated need not be optimal. Gate level ATPGs such as GATEST[17] and STRATEGATE[18–20] have successfully utilized GAs to generate high-quality vectors for stuck-at fault coverage. In [12], GA has been employed in addition to abstraction guided simulation in to reach corner states at the gate level. In [5], GA has been used at the RTL design validation for generating vectors which maximize the statement coverage in the HDL description. A key factor attributed to the success of any GA is the diversity of the population in any given iteration, which is achieved by having a diverse set of fitness values. Therefore, the fitness function, used to assign fitness values, should be able to differentiate between individuals with similar properties.

In our work, each individual is an array of bits of fixed length. All individuals in the initial population are initialized with randomly-generated bit-arrays. The length of the individual is determined such that it provides visibility into the subsequent cycles. In *selection*, two individuals are randomly picked, and the individual with the higher fitness is selected. After 2 individuals are selected as parents, they undergo crossover. In crossover, the offsprings are created by copying the parents' bit arrays and swapping the bits of the offsprings with

a swap probability of 0.5. This technique is called *uniform crossover*. The individuals are then mutated by flipping one of the bits in the array, with a probability close to 1:10^5.

2.4 Verilator

Verilator [8] is an open source tool which converts synthesizable Verilog code into C++ code. More specifically, it compiles the RTL code into a much faster optimized model, while maintaining the same functionality. The optimized model is then encapsulated in a C++ module and the RTL circuit itself is exposed to the user as a C++ class. Verilator also allows the user to instrument the C++ code with unique counters which can be used to compute metrics such as branch or toggle coverage.

During the preprocessing step of our test generation technique, the RTL circuit description written in Verilog is source-to-source compiled into C++ using Verilator. The C++ code obtained will be henceforth referred to as "Verilated C++" code. Verilator converts conditional constructs such as *if, else-if, else* and *case* statements in Verilog to multiple nested if-else blocks in C++. Else-if branches in the Verilog code are converted to if blocks nested inside an else. Similarly, each block in a case statement is converted to an if condition, and the whole case statement is represented as a nested if-else block as shown in Fig. 2.1a, 2.1b.

In addition, each basic block in the C++ code corresponds to a branch in the Verilog code. Basic blocks and branches are used interchangeably in this thesis. During this compilation, instrumentation is added for each branch in Verilog by adding a unique branch counter for the corresponding basic block in the C++ code. Each time the basic block is executed, the corresponding counter is incremented. Comparing the branch counter values in the current and previous cycle provides a trace of all branches that were reached in the current cycle. The instrumented C++ code is then compiled into a static library, which exposes the circuit

```
...

case (st)

    0:  begin
            st = 1;
            cnt = 4;
            ko <= 0;
    end

    1:  begin
            if (cnt == 0) begin
                cnt = 7;
                st = 2;
            end
            else begin
                cnt = cnt - 1;
                st = 1;
            end
        end

    2:  begin
            if (cnt == 0)
                st = 0;
            else if (key == 1) begin
                ko <= 1;
                st = 3;
            end
            else begin
                st = 2;
                cnt = cnt - 1;
            end
        end

    3:  st = 4;

    4:  st = 3;

    default: st = 0;

endcase

...
```

(a) Verilog

```
if ((0 == st)) {
    ++(__Vcoverage[0]);
    st = 1;
    cnt = 4;
    ko = 0;
} else {
    if ((1 == st)) {
        ++(__Vcoverage[3]);
        if ((0 == cnt)) {
            ++(__Vcoverage[1]);
            cnt = 7;
            st = 2;
        } else {
            cnt = (7 & (cnt - 1));
            ++(__Vcoverage[2]);
            st = 1;
        }
    } else {
        if ((2 == st)) {
            ++(__Vcoverage[7]);
            if ((0 == cnt)) {
                ++(__Vcoverage[4]);
                st = 0;
            } else {
                if (key) {
                    ++(__Vcoverage[5]);
                    st = 3;
                    ko = 1;
                } else {
                    cnt = (7 & (cnt - 1));
                    ++(__Vcoverage[6]);
                    st = 2;
                }
            }
        } else {
            if ((3 == st)) {
                ++(__Vcoverage[8]);
                st = 4;
            } else {
                if ((4 == st)) {
                    ++(__Vcoverage[9]);
                    st = 3;
                } else {
                    ++(__Vcoverage[10]);
                    st = 0;
                }
            }
        }
    }
}
```

(b) Verilated C++

Figure 2.1: Example Verilog code and Verilator C++ code

as a C++ class. This class also contains the database of branch counters which is accessible at run-time.

Chapter 3

Branch-guided Functional Test Generation for RTL

3.1 Introduction

With the increasing complexity of modern-day processors and system-on-a-chip, more than 50% of the design efforts are invested in validating the chip. In order to meet time-to-market, there is a need for automated techniques to generate intelligent functional vectors.

In recent years, formal verification methods such as bounded model checking (BMC) and symbolic execution have been used for generating functional tests. In [6, 34] the RTL design is simulated to generate a concrete trace. Using the concrete trace, an RTL symbolic execution path is extracted and one or more conditional expressions can be negated to reach the targeted path (or branch) in the circuit. The path with the negated symbolic expression is then passed to a Satisfiability Modulo Theory (SMT) solver to obtain satisfiable input assignments for the target path. Although such techniques can be used to reach every corner

case in theory, the number of input vectors generated are limited due to the computational cost of each call to the SMT solver. Hence, the paths (or branches) which require longer input sequences are not reached, since the circuit would have to be unrolled that many cycles. Therefore, the de facto approach to validate larger designs is simulation. However, navigating in large state spaces to reach hard corners remains a tremendous challenge for simulation-based test generators.

In this work, we propose a novel functional test generation method which incorporates intelligence from the design to reach hard corner branches. The RTL design is first cross-compiled into C++ and instrumented for branch coverage using Verilator[8]. Our method has a two-stage execution process. In the first stage, the goal is to generate a minimal sequence of vectors which have maximal branch coverage. In the second stage, all branches which have been reached are removed, and we target only the remaining unreached branches. Genetic Algorithms (GAs) are used to navigate the search. GA has been found to be effective in converging to an effective solution in large spaces, and is more scalable than deterministic techniques [35].

In order to assist the GA on reaching hard branches, as a preprocessing step, the instrumented C++ code is parsed and a light-weight static analysis is performed to obtain certain semantics in the circuit, such as flow of control and the hierarchy between the basic blocks. We also propose a few data structures which efficiently model these semantics of the RTL design in terms of branches.

3.2 Related Work

Several approaches have been proposed for test generation at the RTL. Corno et al. proposed a GA-based ATPG in [5], where the input RTL description is first instrumented and then

simulated using a commercial simulator to produce an execution trace. The underlying GA-based procedure then interacts with the simulator via the trace files, and tries to generate an input sequence which executes a target statement or branch in the RTL code which has not yet been reached. While trying to achieve maximum statement coverage, the algorithm is unable to distinguish between easy and hard branches. Therefore, many of the hard-to-reach corner cases remain unreached.

HYBRO is a coverage-driven technique, proposed in [6], which combines dynamic and static analysis to generate high coverage input vectors. The RTL code is instrumented and an execution trace is obtained by a concrete simulation. From the trace, the symbolic expression and the initial guard branches are extracted. The guards are then mutated and fed to an SMT solver to obtain satisfiable input assignments. This approach allows HYBRO to effectively explore the design and achieve high branch coverage, but would require the control-flow graph to be unrolled several times before the target branch can be reached. Such symbolic execution may lead to path explosion, and limits the number of cycles that can be unrolled. Therefore, the branches which require longer input sequences are not reached.

On the other hand, BEACON[7] is a simulation-based design validation technique which uses a marriage of ant colony optimization and evolution as a heuristic to achieve high branch coverage. The RTL code is instrumented with unique counters for each branch using Verilator[8] to help determine the branches which have been reached for a given input sequence. These input sequences resemble ants traversing various parts of the circuit and depositing pheromones. A high counter value for a particular branch is analogous to excess pheromones being deposited on the branch because it is reached by multiple ants. Such branches are considered easy to reach, and are trimmed from the search space so that the ants can target the harder branches. To make faster and better decisions, a hybrid extension to BEACON was proposed in [32]. As a pre-processing step, the control-flow graph (CFG)

of the RTL design is extracted. Then, a frontier search is invoked periodically using an SMT-based BMC which helps navigate the narrow search paths that might be missed with random search.

3.3 Test Generation Framework

This section describes the high-level procedure of our approach. In the pre- processing step, we use Verilator to cross compile the RTL circuit (in Verilog) into C++ and instrument the branches. A light-weight static analysis of the Verilated C++ code is performed to extract additional information from the circuit. The procedure then initiates a two-stage GA-based heuristic search targeting maximum branch coverage. Before we go into the details of the Genetic Algorithm, we describe the various data structures proposed to augment the GA.

The Verilated C++ code comprises of nested if-else statements, which inherently form a structured relation amongst the branches. The Verilated C++ code is statically analyzed to extract these structured relations between the branches both within a single cycle and across consecutive cycles. This static analysis can be broadly divided into the construction of the following data structures:

3.3.1 Branch Hierarchy Tree

In the Verilated C++ code listed in Fig. 3.1b, it is seen that several branches are nested inside other branches. Since the outer branch must be reached first in order to reach the inner branch, there exists a directed relation between the two branches. This relation is also acyclic and transitive, since the innermost branch can be reached only if the outer branches have been reached first. This hierarchy within a single cycle is best modeled by a tree where

each branch in the C++ code constitutes a node in the tree. The root of the tree is a null node which does not correspond to any branch. There exists a directed edge (A, B) from node A to node B if and only if basic block B is fully enclosed inside basic block A. In other words, A is a parent branch which must always be reached first in order to reach the child branch B. This tree is called the branch hierarchy tree (BHT). The BHT for the code listed in Fig. 3.1b is shown in Fig. 3.2. Since the relation is transitive, only the smallest basic block is considered for the parent branch. This helps us avoid the transitive edges, and ensures that every child branch has a unique parent branch. Every branch in Verilog has a corresponding branch in C++ after the translation. Verilator also assigns a unique index to each of these branches. But the reverse is not true because of the way Verilator performs the translation. Therefore, nodes in the BHT which have a corresponding branch in Verilog are labeled with their unique index, while the other nodes are unlabeled.

For example, in the code listed in Fig. 3.1b, the basic block with branch index 4 is enclosed inside the basic block with index 7. Therefore, a directed edge exists from node 7 to node 4 in the example BHT. Similarly, the basic blocks with indices 5 and 6 are fully enclosed in an else block, which in turn is enclosed in the basic block 7. Therefore, there is a directed edge from node 7 to the unlabeled *else* node, and 2 more directed edges from the *else* node to nodes 5 and 6. The *else* statement does not have a corresponding construct in the Verilog code and was added by Verilator during the translation. Therefore, it doesn't have a unique branch index and its corresponding node in the BHT is not labeled.

A binary branch hierarchy tree (i.e., every non-leaf node has 2 children) exhibits the following properties:

1. Exactly one leaf node is traversed in any given cycle.

2. By construction, there exists exactly one path from the root node to the leaf node.

```
...

case (st)

    0:  begin
            st = 1;
            cnt = 4;
            ko <= 0;
    end

    1:  begin
            if (cnt == 0) begin
                cnt = 7;
                st = 2;
            end
            else begin
                cnt = cnt - 1;
                st = 1;
            end
        end

    2:  begin
            if (cnt == 0)
                st = 0;
            else if (key == 1) begin
                ko <= 1;
                st = 3;
            end
            else begin
                st = 2;
                cnt = cnt - 1;
            end
        end

    3:  st = 4;

    4:  st = 3;

    default: st = 0;

endcase

...
```

(a) Verilog

```
if ((0 == st)) {
    ++(__Vcoverage[0]);
    st = 1;
    cnt = 4;
    ko = 0;
} else {
    if ((1 == st)) {
        ++(__Vcoverage[3]);
        if ((0 == cnt)) {
            ++(__Vcoverage[1]);
            cnt = 7;
            st = 2;
        } else {
            cnt = (7 & (cnt - 1));
            ++(__Vcoverage[2]);
            st = 1;
        }
    } else {
        if ((2 == st)) {
            ++(__Vcoverage[7]);
            if ((0 == cnt)) {
                ++(__Vcoverage[4]);
                st = 0;
            } else {
                if (key) {
                    ++(__Vcoverage[5]);
                    st = 3;
                    ko = 1;
                } else {
                    cnt = (7 & (cnt - 1));
                    ++(__Vcoverage[6]);
                    st = 2;
                }
            }
        } else {
            if ((3 == st)) {
                ++(__Vcoverage[8]);
                st = 4;
            } else {
                if ((4 == st)) {
                    ++(__Vcoverage[9]);
                    st = 3;
                } else {
                    ++(__Vcoverage[10]);
                    st = 0;
                }
            }
        }
    }
}
```

(b) Verilated C++

Figure 3.1: Example Verilog code and Verilator C++ output

3. The branches traversed in the given cycle correspond to the set of labeled nodes on the unique path from the root node to the leaf node.

Therefore, if all leaf nodes branches in the BHT have been reached, then every path in the tree has been traversed at least once. This means that all the indexed branches in the C++ code would also have been reached. Therefore, we can conclude that all branches in the Verilog code would have also been reached, reaching the maximum branch coverage.

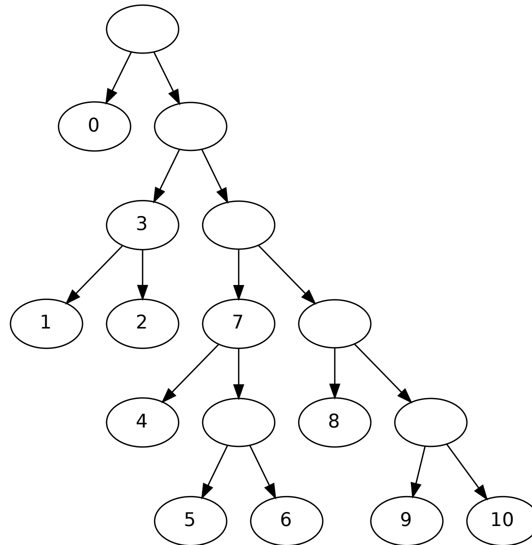


Figure 3.2: Branch Hierarchy Tree

3.3.2 Assignment Table

Every branch in the Verilated C++ code is associated with a condition. This condition is called the *activating condition* for the branch. Since the target branch can be nested inside other branches, as seen in the previous discussion, we would need to satisfy the activating conditions of all the parent branches in order to reach the target branch. The conjunction of the activating conditions of all parent branches is called the *preceding condition* for the target branch. Therefore, each branch is associated with 3 values: (1) a unique index obtained from Verilator, (2) an activating condition and (3) a preceding condition. For each signal present in the activating condition, we extract the list of all assignments to that signal in the entire code. Each assignment is identified by the unique index of the branch in which it occurs. Next, the assignments are added to an assignment graph. Then, each assignment that was added is checked for references to other signals, and added as a new level to the assignment graph. This is continued for a fixed number of levels or until fixed point is reached.

Each path in the assignment graph is then formulated as an SMT expression. If the assign-

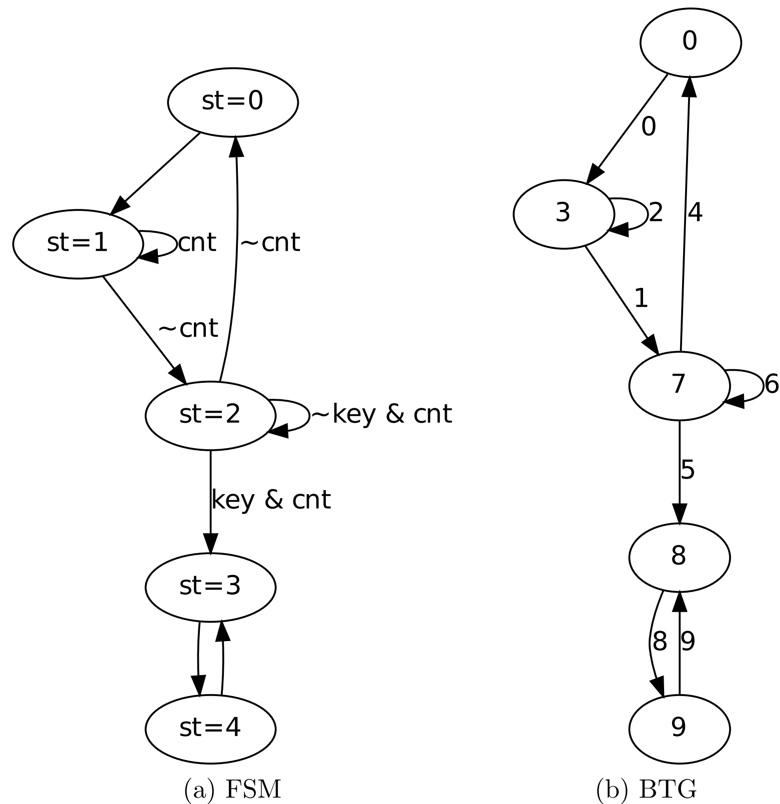
```
0 S (4, 10)
1 S (2, 6)
2 S (2, 6)
3 S (0, 2)
4 S (2, 6)
5 S I
6 S I
7 S (1, 6)
8 S (5, 9)
9 S (8)
10 S
```

Figure 3.3: Assignment Table

ment is satisfiable, we can narrow down the list of potential branches which must be reached before the target branch can be reached. This list of potential branches is computed for each branch, and an assignment table is generated. An example of the assignment table for the code listed in Fig. 3.1b is shown in Fig. 3.3.

In this example, to satisfy the activating condition of branch 7, either branch 1 or 6 should be reached in the previous cycle. This analysis can be recursively applied to branches 1 and 6, and so on. In another example, the activating condition of branch 5 can be satisfied directly from the primary inputs, as indicated by 'I'. Finally, if the assignment table entry for a branch is empty, then that branch is unreachable, as there are no other branches or input combinations which satisfy its activating condition. In the assignment table listed in Fig. 3.3, the entry for branch 10 is empty. Therefore 10 is unreachable.

The assignment table was first proposed in [36] to prove the reachability of branches in the RTL using signal domain analysis. The assignment table is primarily used for checking the reachability of a given branch. In this work, we use the assignment table in conjunction with the branch hierarchy tree to generate a structure called a branch transition graph, which is explained next.



(a) FSM (b) BTG

Figure 3.4: Finite State Machine and Branch Transition Graph

3.3.3 Branch Transition Graph

Finite state machines (FSMs) are generally expressed as case statements in Verilog, and each state is denoted by a case block. Fig. 3.4a represents the FSM denoted by the Verilog code listed in Fig. 3.1a. The case statement in Verilog is converted into nested if-else statements in C++ by Verilator. Each case block in Verilog corresponds to an if statement in C++, and the entire case statement is then represented as a nested if-else- if ladder as shown in Fig. 3.1b. The if-else-if ladder corresponding to a case statement has a well-defined structure in the BHT. Fig. 3.2 shows the BHT for the example FSM above. The if conditions that correspond to the case blocks are assigned a unique branch index by Verilator, while the else statements are not assigned any index, as they exist only in the C++ code. Therefore, a case

statement in Verilog resembles a recursive chain of unlabeled nodes in the BHT, where each unlabeled node has one labeled child node and one unlabeled child node. The last unlabeled node in the chain has two labeled child nodes. In the example shown in Fig. 3.2, the nodes labeled 0, 3, 7, 8, 9 and 10 correspond to the case-blocks and their parents form the chain of unlabeled nodes. The labels obtained from these labeled child nodes correspond to the branch indices of the case-blocks in the Verilog code.

From the BHT, which depicts the relations of branches in a single cycle, we can construct another structure, namely the branch transition graph (BTG), that relates the flow of transitions across cycles. To construct this graph, we first extract the branch indices which correspond to the case-blocks, by the method mentioned above. These branch indices form the set of nodes in the branch transition graph.

For every leaf node with branch index e in the BHT, we add an edge, with the same label e, from node i to node j in the branch transition graph if and only if

1. Either $e = i$ or $e \in$ descendants(i) in the BHT, and
2. $e \in$ assignment table entry for j.

For the example BHT shown in Fig. 3.2, the branch indices $(0, 3, 7, 8, 9, 10)$ correspond to the branch indices of the case-blocks. These form the set of nodes in the BTG shown in Fig. 3.4b. We observe that 0 is a leaf node in the BHT. Therefore, it satisfies condition (1). 0 also appears in the assignment table entry for branch 3. Therefore, we add a directed edge labeled 0 from node 0 to node 3 in the BTG. As another example, we see that the leaf node 5 in the BHT is a descendant of branch 7. Branch 5 also appears in the assignment table entry for branch 8. Therefore, we add a directed edge labeled 5 from node 7 to branch 8. The BTG is further refined to accommodate special cases such as unreachable branches. If a branch u is deduced to be unreachable from the assignment table, and a node or edge with

label u exists, then it is removed from the BTG. In the example shown in Fig. 3.4b, no node or edge with label 10 exists because the branch is deduced to be unreachable.

Using any graph-traversal algorithm, we can compute the paths from branch S to a target branch T in the BTG. A path from S to T, $path(S, T)$, is an ordered list of edges in the BTG without including self-loops present in the path. For example, $path(0, 8) = (0, 1, 5)$ and $path(7, 3) = (4, 0)$.

3.3.4 Abstract state

In a sequential digital circuit, the contents stored in the memory elements is collectively referred to as the circuit's *state*. In the RTL description, a state S is composed of different signals whose values are stored in registers. A *control-flow* signal is defined as a signal which appears in one or more activating conditions. A control flow signal can be either an input or a register. In this work, we define an *abstract state* as a collection of register that hold the control-flow signal values. This abstract state is analogous to an incompletely specified state in which only the control-flow signals are fully specified and the other registers are don't-cares. For the example shown in Fig 3.1b, the abstract state can be denoted by $st, count, key$.

An abstract state represents a control states in the RTL circuit. In Fig. 3.1b, we can see that the activating condition for branch 2 is $(st == 1) \wedge (cnt > 0)$. The corresponding abstract states which reach branch 2 are $((st == 1) \wedge (cnt \in \{1, 2, 3, 4\}))$. So, we see that a single branch can corresponds to multiple abstract states. Therefore, abstract states provides a more fine-grained representation of control states than branches, and play an important role in determining whether a generated vector reaches new control states.

3.4 Genetic Algorithm Search

The GA-search is divided into 2 stages, as described subsequently. The previously constructed BHT and BTG are used to guide the GA towards maximum branch coverage.

3.4.1 GA: Stage 1

The goal of stage 1 is to reach as many easy branches as possible, while keeping the number of vectors minimal. The initial state S_0 is the circuit state after the circuit has been initialized or reset. This stage is further subdivided into multiple rounds. In each round, the population is initialized with random vectors and the start state S_0. Then, the population is evolved over several generations to produce an individual which is added to the test sequence. The GA terminates if

1. The number of generations has reached the maximum limit or,
2. No new control state has been reached in the current generation or,
3. The maximum branch coverage by any individual in the current population is lower than that of the previous generation.

For every state encountered, the abstract state is generated and stored in a database. The fitness of an individual is measured as a function of the number of new branches the individual has reached in the current round. Finding new or rarely traversed branches are favored since they might lead us to new uncovered branches. But, a hard-to-reach branch might be reached only after an easy branch has been reached multiple times. Therefore, we do not distinguish between hard and easy branches by the number of times a branch has been reached. In addition to branches, newer control states are also given higher fitness values.

Once the GA terminates, the individual with the highest fitness is obtained and its coverage is computed. Next, we compute the smallest index of the vector in the individual that achieved the observed coverage. This index is called the $index_{max}$. All the vectors until $index_{max}$ are then added to the test sequence while the remaining vectors discarded. Similarly, all the states reached by the individual until $index_{max}$ are also added to a global state database. The state corresponding to the vector with $index_{max}$ is set as the start state S_0 for the next round. The rounds in Stage 1 terminate if

1. The maximum number of rounds is reached, or
2. The fittest individual added to the test set does not reach any new branch and the start state, S_0, for the next round has been encountered previously.

The parameters that affect the fitness function in Stage 1 are shown in Section. 3.4.3.

At the end of Stage 1, the easy-to-reach branches are filtered from the hard ones, thus narrowing the targeted search in the next stage. The test sequence and the final state reached in Stage 1 are then provided as inputs to Stage 2.

3.4.2 GA: Stage 2

After Stage 1 terminates, the branches reached by the last input vector in Stage 1 is obtained. From the list of branches, we compute the edge in the BTG which was taken in the last cycle of Stage 1. The node on which the edge is incident upon is called the *start node*. We also compute the list of all branches which have not been reached in Stage 1, and are leaf nodes in the BHT, namely the *uBranchList*. We then compute the paths from the *start node* to every other branch in the *uBranchList* by a breadth first traversal of the BTG. The branch corresponding to the node with the shortest path is chosen as the *target branch*.

The individuals which stay on the path are given a higher fitness value, when compared to individuals that deviate from the path. A transition is considered to be on the target path, if (1) it is an edge on the target path or, (2) if the node reached after the transition in the BTG is on the target path or, (3) if the edge is a self loop in the BTG taken. When an individual deviates from the target path, the remaining vectors are discarded. Contrary to Stage 1, we prefer individuals with longer lengths and are given higher fitness values.

If none of the individuals reach the *target branch*, then the round is terminated and the branch is added back to the $uBranchList$. If the *target branch* has been reached, then the branch is removed from the $uBranchList$. All branches which have been reached incidentally are removed from the $uBranchList$, and the vectors from the individual with the highest fitness is added to the test sequence. In the starting of the next round, the *start node* and the *target branch* are recomputed and the procedure is repeated again. Stage 2 is terminated if (1) The number of rounds has reached the maximum limit or, (2) All the branches have been reached or, (3) Timeout occurred because no new branches have been reached for the last T cycles.

As an example, we look at the code listed in Fig. 3.1b. The branches reached in the last cycle of Stage 1 were (4, 7) and branches 5, 8 and 9 have not been reached. From the BTG in Fig. 3.4b, we see that the corresponding edge taken is labeled 4, and goes from node 7 to node 0. Since edge 4 is incident on node 0, the *start node* is set as 0. $uBranchList$ is populated with the branch indices 5, 8 and 9. Next, the paths for each branch in the $uBranchList$ is calculated, and 5 is found to be the shortest path. Therefore, the *target branch* is set as 5, and the *target path* is the ordered list of edge indices (0, 1, 5). An individual is considered to be on the target path if it reaches the edges 0, 1 and 5. In addition, individuals reaching the branches 2 and 6 are also considered to be on the path, since they are self-loops at nodes 3 and 7 respectively. In this case, an individual is considered to deviate from the path if it is

currently at node 7, and takes the edge labeled 4 instead of 5 or 6. If any individual in the population reaches the *target branch*, then the round is terminated and the new *start node* is set as 8, and 5 is removed from *uBranchList*; else, the new *start node* is set as 0. The next round is initiated with the new *start node*.

The parameters that affect the fitness function in Stage 2 are described in Section 3.4.3.

3.4.3 Fitness functions

The parameters that affect the fitness function in Stage 1 are given below.

f_1 : *Number of branches covered by the individual*

f_2 : *The avg. number of branches covered by the population*

f_3 : *The number of new branches covered by the individual*

f_4 : *The individual's $index_{max}$*

f_5 : *Length of the individual*

f_6 : *Number of unique states reached by the circuit*

For Round 0: $fitness = (f_1 - f_2) * 150 + (f_5 - f_4) * 10 + f_6 * 10$

For Round i, > 0 : $fitness = (f_1 - f_2) * 150 + f_3 * 1500 + (f_5 - f_4) * 10 + f_6 * 10$

The parameters that affect the fitness function in Stage 2 are given below.

b_1 : 1 *if target branch was taken*, 0 otherwise

b_2 : 1 *if target node was reached*, 0 otherwise

b_3 : 1 *if self loop was taken*, 0 otherwise

b_4 : 1 *If branch taken deviates from path*, 0 otherwise

$fitness = 2000 * b_1 + 1000 * b_2 + 500 * b_3 - 1000 * b_4$

3.5 Results

The whole algorithm was implemented in C++ on an Intel Xeon 3GHz workstation with 6GB memory running Ubuntu 12.04. The assignment table was generated using Python 2.7.5 and the z3 [37] SMT solver (Python version). Experiments were conducted on a number of ITC99 circuits [38] as well as 4 modules from the OpenRISC 1200 circuit from Open Cores[39]. The 4 OR1200 modules are the (1) instruction cache controller, (2) data cache controller, (3) wishbone bus interface and the (4) exception handler, and are referred to as OR1200-x where x is the module index listed above. The population size is set to 256 for circuits with input size < 5, 512 for input size < 10 and 1024 for input size > 10. The number of generations is set to 8 in Stage 1 and 4 in Stage 2. The length of the individual is set as K_0 times the input size of the circuit, where $K_0 \in [5-20]$ for majority of the circuits. The maximum number of rounds is circuit-dependent, varying from 20-200 rounds for Stage 1 and 100-5000 rounds for Stage 2.

The characteristics of the benchmark circuits are shown in Table. 3.1. Columns 2-4 show the total number of branches, number of input bits and the number of FFs (memory elements) in the circuit. Columns 5 and 6 compare the number of unreachable branches deduced by BMC in [9] and by the assignment table used in our method. Column 7 shows the one-time cost for static analysis, of which construction of assignment table takes majority of the time. The construction of the assignment table is much slower because it uses Python interfaced with the Python version of Z3. By switching to C++ in both cases, we can achieve higher speed-up.

Table 3.2 compares the branch coverage of BEACON [9], HYBRO [6] and our method. It also reports the number of vectors and test execution time for our method and BEACON. Columns 2-4 compare the branch coverage between HYBRO, BEACON and the proposed

Bench	#Branches	#PIs	#FFs	# Unreachable Branches		
				BMC[9]	Assn Table	Time(s)
b01	26	2	3	0	0	8.1
b06	24	2	3	1	1	6.1
b07	20	1	171	1	1	7.9
b10	32	11	14	1	1	7.7
b11	33	7	48	1	1	15.2
b12	105	5	115	1	2	62.2
b14	211	32	398	2	12	166.9
OR1200-0	18	111	75	0	1	4.72
OR1200-1	24	144	77	0	1	5.80
OR1200-2	19	111	4	0	0	3.66
OR1200-3	47	171	96	0	0	14.02

Table 3.1: Benchmark Characteristics

approach. Columns 5-6 compare the number of vectors needed to reach the reported coverage, and their ratio is shown in column 7. The number of vectors needed by HYBRO is not shown in the table, as it has not been reported for most circuits in [6]. Also, since the number of vectors are not reported in [9], column 5 shows the number of vectors reported in [7]. The execution times of BEACON [9] and our approach, reported in columns 8 and 9, are very similar.

Bench	Branch Coverage %			#Vectors			Time(s)	
	HYBRO	BEACON	GA	[7]	GA	Redn Ratio	[9]	GA
b01	94.44	**100**	**100**	113	**21**	5.38	**0.002**	0.04
b06	94.12	**100**	**100**	1731	**20**	86.55	**0.02**	0.06
b07	NA	**95.00**	**95.00**	759	**47**	16.14	0.41	**0.06**
b10	96.77	**100**	**100**	3547	**136**	26.08	3.12	**0.7**
b11	91.30	**96.88**	**96.88**	1235	**619**	2.0	4.2	**2.34**
b12	NA	97.1	**100**	37006	**34880**	1.06	69.6	**16.42**
b14	83.50	93.4	**97.63**	4381	**356**	12.31	94.2	**22.87**
OR1200-0	93.75	89.47	**94.44**	642	**23**	27.91	2.82	**0.22**
OR1200-1	96.30	92.00	**100**	2146	**25**	85.84	2.75	**0.38**
OR1200-2	**100**	**100**	**100**	1261	**11**	114.64	2.82	**0.20**
OR1200-3	96.61	**97.87**	**97.87**	4615	**499**	9.25	6.4	**5.37**

Table 3.2: Branch Coverage and Comparison with Prior Work

The proposed approach significantly increases the branch coverage in several circuits by using the assignment table to deduce many branches to be unreachable under non-faulty circuit behavior. For example, in b14, the coverage is increased by 4.2% as 12 branches are deduced to be unreachable. The number of vectors needed in the proposed approach to reach the same (if not higher) coverage is up to two magnitudes lower that BEACON. In OR1200-2, only 11 vectors are needed as opposed to 1731 in the case of BEACON. Similarly, $>$ 80x reduction is obtained in case of OR1200-1 and b06. Additionally in b12, the proposed branch transition graph is very effective in guiding the search through the narrow paths, and is able to achieve 100% branch coverage by reaching all the hard-to-reach branches.

Chapter 4

A New Metric for Functional Test Generation

4.1 Introduction

The design complexity of processors and system-on-chip (SOC) has continued to increase over the years as we can afford to fit more transistors on a single chip. Many of these chips find themselves in mission-critical systems such as automobiles, health-care, etc. Thus, ensuring that they are void of bugs and defects is a must. It is estimated that validation consumes up to 50% of the design efforts and is expected to continue to rise [2]. Simulation dominates in today's validation practices, and as such, effective test patterns are needed so that all parts of the circuit can be exercised.

Successful validation of the design requires an effective suite of test inputs (or vectors). For example, random stimuli can cover a portion of the design space, but many hard to reach areas and corner cases remain unexercised. Because corner cases may not be easy to describe

or quantify, metrics that try to correlate the effectiveness of a test suite and the extent to which the design has been exercised have been proposed. In this regard, software-based code coverage metrics, such as branch coverage, state coverage, path coverage, statement coverage, etc. have been used. Unlike software, the RTL description of the hardware succinctly describes one cycle of its operation. Therefore, branch coverage has been a popular and practical metric used because maximum branch coverage implies that all valid control states have been reached in the RTL design.

Although software based metrics such as branch coverage have been widely used in design validation, the degree of correlation with catching defects or bugs is not clear. For example, is 100% branch coverage sufficient? If not, what other practical metrics can be used or combined with branch coverage? Defects in silicon are typically modeled as faults at the gate-level. In [32], it is shown that 100% branch coverage in the RTL code does not translate to 100% stuck-at fault coverage at the gate-level. The results also report that two vector sequences with 100% branch coverage might vary widely in stuck-at coverage. The test suite with a lower gate-level coverage may miss some corner case bugs and defects. Thus, one could reason that adding gate-level coverage (such as stuck-at) could enhance the metric. However, fault simulation at the gate-level is significantly more expensive than at the RTL, and hence something different would be preferable. This motivated us to formulate a metric that can achieve high coverages at both the RTL (branches) and gate-level (stuck-at faults) without resorting to gate-level simulation.

Since RTL allows for arithmetic and relational operations in addition to logical operations, multiple state values might correspond to the same branch. The proposed metric uses this state-level information in addition to the branches, and hence called *state-augmented branch score*. Unlike branch and fault coverage, the new metric is expressed as an absolute score, instead of a percentage value. Test sets that achieve a high score on this metric will achieve a

high branch coverage as well as state coverage. Moreover, stimuli that achieve higher scores on this new metric can also be applied at various levels of abstraction, including pre-silicon validation and non-scan post-silicon validation.

Using this new metric, the test generator attempts to generate effective stimuli using the test framework presented in Chapter 3. The test generation flow is as follows. The RTL code is first cross-compiled into C++ and instrumented for branch coverage using Verilator [8]. Simulation in C++ is significantly faster than in Verilog, and thus helps reduce the overall cost. To assist in reaching hard branches, we used the data structures facilitated by the test framework to model circuit semantics. The proposed metric can effectively distinguish two vector sequences with identical branch coverage at the RTL. The test generation method is also successful in generating high quality vectors which achieve high coverages at both the RTL and gate-level. We validate the effectiveness of the proposed approach on both the branch coverage and gate-level stuck coverage. That is, using the proposed method, we hope to achieve both high branch coverage at the RTL and high stuck-at coverage at the gate-level. In so doing, defects and bugs at either level can be captured.

4.2 Motivation

In [32], it was shown that test sets that achieve similar coverages (close to 100%) may not achieve similar gate-level coverages. In fact, up to 12% difference in fault coverage was observed. Therefore, we conjecture that reaching maximum branch coverage is desirable, it is insufficient to exercise all signals and propagate their effects to an output. To understand the reasoning behind this, let us look at the example RTL code snippet in Fig. 4.1.

The condition to reach branch 1 is $(state == 3) \wedge (count > 8)$. Although the first clause, $(state == 3)$, is satisfied by a single value of $state$, the second clause, $(count > 8)$, can be

```
reg    [3:0] d_in
reg    [2:0] state
reg    [3:0] count

    ...

    if (state == 3'd3) begin    // 3
        if (count > 8)  // 1
            state = 3'd3;
                ...
        else            // 2
            state = 3'd4;
                ...
    end

    ...

    else if (state == 3'd5) begin   // 6
        if (d_in != count)  // 4
            ...

        else                // 5
            ...

    end
```

Figure 4.1: Example Verilog RTL code snippet

satisfied by multiple values. So, two vectors sequences may reach branch 1, but the values assumed by *count* need not be the same. Therefore, the faults excited and/or propagated, and eventually detected can be different even if the branches traversed are the same. Similarly, the condition to reach branch 4 is $(state == 5) \wedge (d_in \neq count)$. Since both sides of the condition $(d_in \neq count)$ are variables, there are multiple ways to satisfy it, and the faults excited and propagated each time can be different. Moreover, in deep corner cases, some branches would need to be visited multiple times.

Therefore, we need a metric in RTL which takes into consideration, the values taken by the state variables when the branches were traversed. Hence, the new metric is called state-augmented branch score.

Based on this new metric, we also propose a new search framework that can efficiently generate the needed vectors. Several data-structures are proposed to facilitate the search, that can help the test generator reason about which execution paths to take in the RTL.

Table 4.1: Comparison of the three metrics

Bench	Branch Cov (%)			Fault Cov (%)			SABS		
	[7]	[32]	[18]	[7]	[32]	[18]	[7]	[32]	[18]
b07	90.00	90.00	90.00	76.26	76.81	76.81	13.75	13.95	13.95
b10	96.88	96.88	96.88	89.13	94.18	92.82	12.5	12.78	12.61
b11	90.91	93.94	93.94	90.20	96.65	96.29	51.58	97.21	77.88
b12	98.09	98.09	87.62	74.92	91.95	53.51	121.99	238.85	77.91
b14	91.47	91.94	91.94	86.75	90.31	91.95	525.64	3671.56	1079.97

4.3 State-augmented Branch Score

In this section, we present the formal definition of the proposed metric. In a sequential digital circuit, the contents stored in the memory elements is collectively referred to as the circuit's *state*. A state is represented as a set of values taken by the flip-flops at the gate level. In the RTL description, a state S is composed of different signals (or state variables) whose values are stored in registers. The state S can be represented as $S = \{s_1, s_2, ..., s_n\}$. For the example shown in Fig. 4.1, the state can be denoted by $state, count, d_in$. Similarly, for the example shown in Fig. 3.1b, the state can be denoted by $st, count, key$. The *state-augmented branch score* (SABS) uses a combination of branch coverage and state values. For each branch, we compute a weight based on the values taken by the state variables when the branch was reached.

For a given test sequence, we can compute the state-augmented branch score using the algorithm shown in Algorithm 4.1. To compute the score, we simulate one vector at a time and save the current state of the circuit. Then, for each branch b_i reached in the current cycle, we store the values of all the state variables into the value set for that branch. Once the simulation is complete, for each branch b_i, we calculate n_{ij}, which is the number of unique values assigned to each variable s_j when branch b_i was reached. This is done by computing the number of unique values in the value set of branch b_i for variable s_j. The weight of the

branch b_i, wb_i is defined as the sum of n_{ij} for all variables s_j. If a branch b_i was not reached, wb_i is assigned a zero score. The state-augmented branch score is defined as the average of all branch scores, given by the closed form expression $SABS = (\sum_{i}^{B} wb_i)/B$, where B is the total number of branches.

Algorithm 4.1 State-augmented branch score computation

```
INPUT : Vector sequence Vec
INIT : value set vSet[:][:] = Φ
for each vector v ∈ Vec do
   Simulate vector v. Compute current state S_v
   for all branches b_i reached by vector v do
      for all variables s_j ∈ S_v do
         vSet[i][j] ← value(s_j)
      end for
   end for
end for
for all branches b_i reached do
   for all variables s_j do
      n_ij = number of unique values in vSet[i][j]
   end for
   wb_i = Σ_j(n_ij)
end for
SABS = Σ_i(wb_i)
RETURN SABS
```

A comparative study of the 3 metrics is shown in Table 4.1 for the different circuits from the ITC99 benchmarks [38]. The three metrics, branch coverage, fault coverage and state-augmented branch score, are computed for the vectors generated by the gate-level ATPG in [18], the RTL functional test generation method in [7] and the mixed-level test generation method in [32] for each circuit. Column 1 shows the name of the benchmark circuit. Columns 2-3 show the number of branch reached by the 2 methods. Columns 4-5 compare the fault coverage achieved by the 2 methods, and columns 6-7 compare the state-augmented branch score for the vectors generated by the 2 methods. It is observed that for most circuits, a vector sequence with higher state-augmented branch score is able to achieve equal (or

higher) branch coverage at the RTL and higher stuck-at fault coverage at the gate-level. However, the fault coverage in the case of b14 cannot be expressed by using branches and state excitations alone as factors. As a consequence, the vectors generated from a gate-level ATPG in [18] can achieve much high stuck-at coverage fault than [32], even though the SABS is much smaller. Thus, we can conclude that the proposed metric is more effective that branch coverage, and a higher score closely translates to higher fault coverage for most circuits.

4.4 Test Generation

After static analysis, the algorithm initiates a branch-guided search which uses Genetic Algorithms (GAs) as a heuristic. The algorithm consists of multiple runs, and each run is further divided into 2 stages. Since 100% branch coverage translates to a high fault coverage at the gate-level, the goal in the first run is to reach the maximum possible branch coverage. At the end of the first run, the SABS is computed and used as the baseline score. For every successive run, the goal is to achieve a better score than the previous run. The search is terminated when the score becomes constant or the maximum run limit is reached.

In the beginning of each run, the circuit is reset and the branch counters in the Verilated C++ code are cleared. Then, a two-stage search for maximum branch coverage is initiated. In the first stage, the search aims to generate a vector sequence with maximum fitness and minimal length. In the second stage, the search is narrowed down to individually target each previously unreached branch one at a time. Since these branches are harder-to-reach, we use the branch hierarchy tree and the branch transition graph to help guide the search. The fitness function used by the GA depends on the current stage of execution.

4.4.1 GA - Stage 1

In the first run of test generation, the goal is to generate a vector sequence to reach maximum branch coverage. In the subsequent runs, in addition to reaching maximum branch coverage, the vector sequence should also achieve a higher SABS. Diversity in the GA population plays a key role in the success of any GA-based approach. To maintain a good diversity, the length of each individual has to been kept short. But most practical circuits require vector sequences which are much longer than the individual length. Therefore, the first stage is divided into multiple rounds.

Since the circuit is reset at the start of each run, the initial state S_0 in round 0 is the circuit state after the circuit has been reset. In each round, the population is initialized with random vectors and the start state S_0. Then, the population is evolved over several generations to produce an individual which is added to the test sequence. The GA terminates if (1) the number of generations has reached the maximum limit or, (2) no new control state has been reached in the current generation or, (3) the maximum branch coverage by any individual in the current population is lower than that of the previous generation. For every state encountered, the abstract state is extracted and stored in a database. Finding new or rarely traversed branches are favored since they might lead us to new uncovered branches. But, a hard-to-reach branch might be reached only after an easy branch has been reached multiple times. Therefore, we do not distinguish between hard and easy branches by the number of times a branch has been reached.

Once the GA terminates, the individual with the highest fitness is obtained and its coverage/score is computed. Next, we compute the smallest index of the vector in the individual that achieved the observed coverage. This index is called the $index_{max}$. All the vectors until $index_{max}$ (including $index_{max}$) are then added to the test sequence while the remaining

vectors discarded. Similarly, all the states reached by the individual until $index_{max}$ are also added to a global state database. The state corresponding to the vector with $index_{max}$ is set as the start state S_0 for the next round. The rounds in Stage 1 terminate if (1) the maximum number of rounds is reached, or (2) the fittest individual added to the test set does not reach any new branch and the start state, S_0, for the next round has been encountered previously.

4.4.2 GA - Stage 2

At the end of Stage 1, all branches which have been reached are dropped from the search space. The list of branches reached by the last input vector in Stage 1 is obtained. From the list of branches, we compute the edge in the BTG which was taken in the last cycle of Stage 1. The node on which the edge is incident upon is called the *start node*. We also compute the list of all branches which have not been reached in Stage 1, and are leaf nodes in the BHT. This list is called *uBranchList*. We then compute the paths from the *start node* to every other branch in the *uBranchList* by a breadth first traversal of the BTG. The branch corresponding to the node with the shortest path is chosen as the *target branch*. The individuals which stay on the path are given a higher fitness value, when compared to individuals that deviate from the path. A transition is considered to be on the target path, if (1) it is an edge on the target path or, (2) if the node reached after the transition in the BTG is on the target path or, (3) if the edge is a self loop in the BTG taken. When an individual deviates from the target path, the remaining vectors are discarded. Contrary to Stage 1, we prefer individuals with longer lengths and are given higher fitness values. If none of the individuals reach the *target branch*, then the round is terminated and the branch is added back to the *uBranchList*. If the *target branch* has been reached, then the branch is removed from the *uBranchList*. All branches which have been reached incidentally are removed from the *uBranchList*, and the vectors from the individual with the highest fitness

is added to the test sequence. At the start of the next round, the *start node* and the *target branch* are recomputed and the procedure is repeated again. Stage 2 is terminated if (1) The number of rounds has reached the maximum limit or, (2) All the branches have been reached or, (3) All branches have reached their maximum try limit.

As an example, we look at the code listed in Fig. 3.1b. The branches reached in the last cycle of Stage 1 were (4, 7) and branches 5, 8 and 9 have not been reached. From the BTG in Fig. 3.4b, we see that the corresponding edge taken is labeled 4, and goes from node 7 to node 0. Since edge 4 is incident on node 0, the *start node* is set as 0. *uBranchList* is populated with the branch indices 5, 8 and 9. Next, the paths for each branch in the *uBranchList* is calculated, and 5 is found to be the shortest path. Therefore, the *target branch* is set as 5, and the *target path* is the ordered list of edge indices (0, 1, 5). An individual is considered to be on the target path if it reaches the edges 0, 1 and 5. In addition, individuals reaching the branches 2 and 6 are also considered to be on the path, since they are self-loops at nodes 3 and 7, respectively. In this case, an individual is considered to deviate from the path if it is currently at node 7, and takes the edge labeled 4 instead of 5 or 6. If any individual in the population reaches the *target branch*, then the round is terminated and the new *start node* is set as 8, and 5 is removed from *uBranchList*; else, the new *start node* is set as 0. The next round is initiated with the new *start node*.

4.4.3 Fitness functions

In GAs, the fitness function is formulated by taking the metric used into account. The metric used in our GA is the proposed state-augmented branch score. For a large population of individuals, computing the number of new unique state values seen by each individual is computationally expensive. Also, the weight of all state variables are not equal, as exploring

states where a control state variables is assigned a new value might be more important than a data-path variable. Therefore, we use a faster metric which utilizes abstract states and hamming distance. Since abstract states represent control state variables, the weight tilts more in the favor of control state variables.

Stage 1 fitness function parameters

b_1 : *Number of branches covered by the individual*

b_2 : *The avg. number of branches covered by the population*

b_3 : *The number of new branches covered by the individual*

v_1 : *The individual's* $index_{max}$

v_2 : *Length of the individual*

s_1 : *Number of new control states reached by the circuit*

s_2 : *Sum of avg hamming distances of previous control states* s_3 : *Number of FFs in the circuit* $numFFs$

For Round 0:

$fitness = (b_1 - b_2) * 150 + (v_2 - v_1) * 10 + 0.4 * (s_1 * s_3 + s_2)$

For Round $i, > 0$:

$fitness = (b_1 - b_2) * 150 + f_3 * 1500 + (v_2 - v_1) * 10 + 0.4 * (s_1 * s_3 + s_2)$

Stage 2 fitness function parameters

s_1 : *Number of new control states reached by the circuit*

s_2 : *Sum of avg hamming distances of previous control states*

b_1 : 2000 *if target branch was taken*, 0 otherwise

b_2 : 1000 *if target node was reached*, 0 otherwise

b_3 : 500 *if self loop was taken*, 0 otherwise

b_4 : -1000 *If branch taken deviates from path*, 0 otherwise

$$fitness = 200 * s_1 + 10 * s_2 + b_1 + b_2 + b_3 + b_4$$

To compute s_1, s_2, we use the following expression:

if(state S unique)

$s_1 = s_1 + 1$

else

$s_2 = s_2 + avg_i(hamming_dist(S, S_{prev}[i]))$

where S_{prev} is the list of all states with same abstract state as S.

4.5 Results

The whole algorithm was implemented in C++ on an Intel Xeon 3GHz workstation with 6GB memory running Ubuntu 12.04. The assignment table was generated on an Intel Core i5-3210M 2.5GHz laptop running Windows 7, using Python 2.7.5 and the z3 [37] SMT solver (Python version). Experiments were conducted on a number of ITC99 circuits [38]. The RTL circuit (written in Verilog) is synthesized into gate-level benchmarks using Synopsys Design Compiler [40], which is used to estimate primarily to estimate the fault coverage of the vectors generated by the proposed test generation method.

The characteristics of all benchmark circuits is shown in Table. 4.2. Columns 1 through 4 show the name, number of primary inputs (PIs), primary outputs (POs) and flip-flops in the RTL description. Columns 5 and 6 report the total number of branches in the RTL code and the number of unreachable branches deduced using the assignment table respectively.

Columns 7 and 8 report the number of equivalent collapsed faults in the gate level circuit and the number of untestable faults reported by STRATEGATE [18].

Bench	#PI	#PO	#FF	#Branches		#Faults	
				Total	Unreach	Total	Untest
b07	3	8	43	20	1	1274	6
b10	12	6	11	32	1	515	1
b11	8	6	25	33	1	1152	19
b12	6	6	115	105	2	2902	7
b14	33	54	253	211	12	11101	1

Table 4.2: Benchmark Characteristics

4.5.1 Algorithmic Settings

The population size is set to 256 for circuits with input size < 5, 512 for input size < 10 and 1024 for input size > 10. The number of generations is set as 8 in Stage 1 and 4 in Stage 2. The length of the individual is set as K_0 times the input size of the circuit, where $K_0 \in [5-20]$ for majority of the circuits. The maximum number of rounds is circuit-dependent, varying from $20-200$ rounds for Stage 1 and $100-5000$ rounds for Stage 2. The maximum try limit for each branch in Stage 2 is set as 100.

4.5.2 Vector Generation

We provide a comparison between vectors generated using 2 metrics, branch coverage and state-augmented branch score (SABS) by using two different fitness functions in our GA-based search. The fitness function for state-augmented branch score (GA-S) uses the same fitness function shown in Section 4.4.3. The fitness function for branch coverage (GA-B) uses the fitness functions shown in Section 3.4.3.

The vectors generated using both the metrics are compared in Table 4.3. Column 1 shows the

Bench	Branch Cov %		SAB Score		Fault Cov %		Num Vectors		Time(s)	
	GA-B	GA-S	GA-B	GA-S	GA-B	GA-S	GA-B	GA-S	GA-B	GA-S
b07	95.00	95.00	13.95	13.95	75.24	**75.79**	88	279	0.06	0.26
b10	100	100	12.56	**12.81**	90.87	**93.60**	406	1835	0.71	1.85
b11	96.96	96.96	69.00	**89.39**	95.23	**96.91**	1012	3053	2.34	7.52
b12	100	100	117.57	**186.94**	85.65	**91.95**	37377	191275	16.42	326.4
b14	97.63	97.63	248.46	**2982.39**	85.65	**90.06**	1160	73491	22.87	1964.2

Table 4.3: Comparison of branch coverage vs SABS

benchmark circuit. Columns 2-3 show the branch coverage achieved by using the 2 metrics. . Columns 4-5 report the state-augmented branch score for the vectors generated using the 2 methods. The fault coverages achieved by the vectors generated using the 2 metrics are reported in columns 6-7. The number of vectors and the execution times are reported in columns 8-9 and 10-11, respectively. We can see that using SABS instead of branch coverage as the metric results in equal branch coverage at the RTL but higher stuck-at fault coverage for all the benchmark circuits.

The vectors generated using SABS are also compared with the vectors generated using the method proposed in [32] in Table 4.4. Column 1 shows the benchmark circuit. Columns 2-3 show the branch coverage of the 2 methods. It is to be noted that the value is obtained after including branches which have been proved unreachable by the assignment table as covered. Therefore, these values might vary from the values reported in [32]. Columns 4-5 report the state-augmented branch score for the 2 methods. The fault coverages achieved by the vectors generated using the 2 methods are reported in columns 6-7. The number of vectors and the execution times are reported in columns 8-9 and 10-11, respectively.

We can see that using SABS instead of branch coverage as the metric results in equal branch coverage at the RTL but higher stuck-at fault coverage for all the benchmark circuits. We also observe that our test generation framework reaches similar or better coverage for hard-to-test circuits like b11 and b12, while maintaining a smaller test vector size. Since the number

Bench	Branch Cov %		SAB Score		Fault Cov %		Num Vectors		Time(s)	
	GA-S	[32]	GA-S	[32]	GA-S	[32]	GA-S	[32]	GA-S	[32]
b07	95.00	95.00	**13.95**	**13.95**	75.79	**76.81**	279	495	0.26	7.4
b10	100	100	**12.81**	12.78	93.60	**94.17**	1835	3174	1.85	1.5
b11	96.96	96.96	89.39	**97.21**	**96.91**	96.65	3053	5796	3.87	7.2
b12	100	100	186.94	**238.85**	**91.95**	**91.95**	191275	327577	326.4	184.0
b14	97.63	97.63	2982.39	**3671.56**	90.06	**90.31**	73491	310000	1964.2	424.3

Table 4.4: Comparison of SABS with Prior Work

of vectors generated by [32] is greater in all circuits, it has a higher chance of exploring a larger set of state values. Therefore, the SABS score for b11 and b12 are much higher when compared to GA-S. Although we previously mentioned that SABS is not a sufficient metric to reach high fault coverage in b14, we see that there is a 5% increase in fault coverage by increasing the SABS by 10-15 times in both GA-S and [32], thus proving that it is effective in achieving higher fault coverage.

Chapter 5

Conclusion

5.1 Conclusion

In this paper, we presented a novel genetic algorithm based method for functional test generation at the RTL. To guide the GA based search, certain semantics of the design were extracted using static analysis. We also proposed effective data structures to model these semantics. The proposed method offers an efficient way for design exploration, since it achieves very high branch coverage (100% for several circuits) with a small number of vectors. Experimental results show that the proposed method reaches equal or higher branch coverage as reported previously. Furthermore, the number of vectors needed is up to two orders of magnitude smaller than those needed by the prior techniques.

In this paper, we presented a novel branch-guided metric for functional test generation called state-augmented branch score (SABS), which can be used to generate test vectors for different levels of abstraction. We also proposed a search framework which can effectively generate these test vectors. Using the new metric and search framework, we were able to achieve

similar or higher RTL (branch) and gate-level (stuck-at fault) coverage, while maintaining smaller test vector sequences for some hard-to-test circuits.

5.2 Future Work

While doing this research, we also came across several ideas which might be worth exploring. This section lists some of these ideas as future work.

1. It is shown in this thesis that, the state augmented branch score is more effective than branch coverage in achieving higher branch coverage and fault coverage. But SABS considers all state variables have equal weights, and the branches are weights based on the number of unique values taken by these state variables. But in most practical circuits, the importance of all state variables are not the same, all branches may not reached with the same ease.

   Some variables might be key in reaching corner states and should be weighted heavier than the others. Similarly, we could add weights to the branches to classify them either as easy or hard to reach, and use this in the metric. Similarly, the correlation between state values and the transition relation between states can also be used as a factor in the proposed metric.

2. During the test generation, we can extract various patterns and relations from the individuals in the GA population and their fitness values. These relations can be used as heuristics to refine the GA search in the subsequent iterations, or even used to provide weights in the SABS computation.

Bibliography

[1] G. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, pp. 82–85, Jan 1998.

[2] H. Foster, "The 2012 Wilson Research Group Functional Verification Study," Aug 2013.

[3] V. V. Acharya, S. Bagri, and M. S. Hsiao, "Branch Guided Functional Test Generation at the RTL," in *Proc. IEEE European Test Symposium (ETS)*, May 2015.

[4] L. T. Wang, Y. W. Chang, and K. T. Cheng, *Electronic Design Automation: Synthesis, Verification, and Test*. Systems on Silicon, San Francisco, CA, USA: Morgan Kaufmann, 2009.

[5] F. Corno, M. S. Reorda, G. Squillero, A. Manzone, and A. Pincetti, "Automatic Test Bench Generation for Validation of RT-level Descriptions: An Industrial Experience," in *Proc. Design, Automation and Test in Europe Conference*, pp. 385–389, 2000.

[6] L. Liu and S. Vasudevan, "Efficient validation input generation in RTL by hybridized source code analysis," in *Proc. Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–6, Mar 2011.

[7] M. Li, K. Gent, and M. S. Hsiao, "Design validation of RTL Circuits using Evolutionary Swarm Intelligence," in *Proc. IEEE International Test Conference (ITC)*, pp. 1–8, Nov 2012.

[8] "Verilator." [Online] http://www.veripool.org/wiki/verilator.

[9] K. Gent and M. S. Hsiao, "Functional Test Generation at the RTL Using Swarm Intelligence and Bounded Model Checking," in *Proc. 22nd Asian Test Symposium (ATS)*, pp. 233–238, Nov 2013.

[10] S. Shyam and V. Bertacco, "Distance-guided hybrid verification with guido," in *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings*, DATE '06, pp. 1211–1216, 2006.

[11] M. Li and M. S. Hsiao, "An ant colony optimization technique for abstraction-guided state justification," in *40th IEEE International Test Conference (ITC)*, pp. 1–10, Nov 2009.

[12] A. Parikh, W. Wu, and M. S. Hsiao, "Mining-guided state justification with partitioned navigation tracks," in *38th IEEE International Test Conference (ITC)*, pp. 1–10, Oct 2007.

[13] W. Wu and M. S. Hsiao, "Efficient design validation based on cultural algorithms," in *Proc. Conference on Design, Automation and Test in Europe (DATE)*, DATE '08, pp. 402–407, 2008.

[14] F. M. De Paula and A. J. Hu, "An Effective Guidance Strategy for Abstraction-Guided Simulation," in *44th ACM/IEEE Design Automation Conference (DAC)*, pp. 63–68, June 2007.

[15] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer Publishing Company, Inc., 2013.

[16] T. Niermann and J. H. Patel, "Hitec: a test generation package for sequential circuits," in *Proc. European Design Automation Conference (EDAC)*, pp. 214–218, Feb 1991.

[17] E. M. Rudnick, J. H. Patel, G. S. Greenstein, and T. M. Niermann, "Sequential Circuit Test Generation in a Genetic Algorithm Framework," in *Proc. 31st Annual Design Automation Conference*, pp. 698–704, 1994.

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

[19] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Dynamic State Traversal for Sequential Circuit Test Generation," *ACM Trans. Design Automation of Electronic Systems*.

[20] 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 *Proc. 10th International Conference on VLSI Design*, pp. 475–481, Jan 1997.

[21] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Application of genetically engineered finite-state-machine sequences to sequential circuit ATPG," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, pp. 239–254, Mar 1998.

[22] A. Giani, S. Sheng, M. Hsiao, and V. Agrawal, "Efficient Spectral Techniques for Sequential ATPG," in *Proc. Conference on Design, Automation and Test in Europe (DATE)*, pp. 204–208, 2001.

[23] A. Giani, S. Sheng, M. Hsiao, and V. Agrawal, "Novel spectral methods for built-in self-test in a system-on-a-chip environment," in *19th IEEE Proc. VLSI Test Symposium (VTS)*, April 2001.

[24] S. Sheng and M. S. Hsiao, "Efficient sequential test generation based on logic simulation," *IEEE Design Test of Computers*, vol. 19, pp. 56–64, Sep 2002.

[25] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Fast static compaction algorithms for sequential circuit test vectors," in *IEEE Trans. on Computers*, vol. 48, pp. 311–322, Mar 1999.

[26] M. S. Hsiao and S. Chakradhar, "State Relaxation Based Subsequence Removal for Fast Static Compaction in Sequential Circuits," in *Proc. IEEE Design Automation and Test in Europe Conference (DATE)*, pp. 577–582, Feb 1998.

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

[28] L. Zhang, I. Ghosh, and M. Hsiao, "Efficient sequential atpg for functional rtl circuits," in *Proc. 18th International Test Conference (ITC)*, vol. 1, pp. 290–298, Sept 2003.

[29] B. Alizadeh and M. Fujita, "Guided gate-level ATPG for sequential circuits using a high-level test generation approach," in *Proc. 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 425–430, Jan 2010.

[30] E. M. Rudnick, R. Vietti, A. Ellis, F. Corno, P. Prinetto, and M. S. Reorda, "Fast sequential circuit test generation using high-level and gate-level techniques," in *Proc. Design, Automation and Test in Europe (DATE)*, pp. 570–576, Feb 1998.

[31] S. Ravi and N. K. Jha, "Fast test generation for circuits with rtl and gate-level views," in *Proc. IEEE International Test Conference (ITC)*, pp. 1068–1077, 2001.

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

[33] J. H. Holland, *Adaptation in Natural and Artificial Systems.* Cambridge, MA, USA: MIT Press, 1972.

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

[35] F. Corno, P. Prinetto, M. Rebaudengo, and M. S. Reorda, "Comparing topological, symbolic and GA-based ATPGs: an experimental approach," in *Proc. International Test Conference*, pp. 39–47, Oct 1996.

[36] S. Bagri, K. Gent, and M. S. Hsiao, "Signal Domain based Reachability Analysis in RTL circuits," in *Proc. International Symposium on Quality Electronic Design (ISQED)*, Mar 2015.

[37] "Microsoft Research z3 SMT solver." [Online] http://z3.codeplex.com.

[38] S. Davidson, "ITC99 benchmark circuits - preliminary results," in *IEEE International Test Conference (ITC)*, p. 1125, 1999.

[39] "Openrisc circuits." [Online] http://www.opencores.org.

[40] "Synopsys Design Compiler." [Online] http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/default.aspx.