Guiding RTL Test Generation Using Relevant Potential Invariants

Tania Khanna

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Michael Hsiao, Chair

Haibo Zeng

Lynn Abbott

June 15, 2018

Blacksburg, Virginia

Keywords: Ant Colony Optimization, Potential Invariants, Branch Coverage, Verilator tool, Daikon tool

Copyright 2018, Tania Khanna

Guiding RTL Test Generation Using Relevant Potential Invariants

Tania Khanna

(ABSTRACT)

In this thesis, we propose to use relevant potential invariants in a simulation-based swarm-intelligence-based test generation technique to generate relevant test vectors for design validation at the Register Transfer Level (RTL). Providing useful guidance to the test generator for such techniques is critical. In our approach, we provide guidance by exploiting potential invariants in the design. These potential invariants are obtained using random stimuli such that they are true under these stimuli. Since these potential invariants are only likely to be true, we try to generate stimuli that can falsify them. Any such vectors would help reach some corners of the design. However, the space of potential invariants can be extremely large. To reduce execution time, we also implement a two-layer filter to remove the irrelevant potential invariants that may not contribute in reaching difficult states. With the filter, the vectors generated thus help to reduce the overall test length while still reach the same coverage as considering all unfiltered potential invariants. Experimental results show that with only the filtered potential invariants, we were able to reach equal or better branch coverage than that reported by BEACON in the ITC99 benchmarks, with considerable reduction in vector lengths, at reduced execution time.

Guiding RTL Test Generation Using Relevant Potential Invariants

Tania Khanna

(GENERAL AUDIENCE ABSTRACT)

Over the recent years, size and complexity of hardware designs are increasing at an enormous rate. Due to this, verification of these designs is of utmost importance and demands much more resources and time than designing of these hardware. To project the information of the designs, developers use Hardware Descriptive Languages (HDL), that includes the important decision points of the system, also called branches of the circuit. There are several methodologies proposed to check how many branches of the design can be traversed by set of inputs. This practice is important to confirm correct functionality of the design as we can catch all the faults in the design at these decision points. Some of these methodologies include checking with random inputs, exhaustively checking for every possible input, investing many hours of labor to verify with appropriate inputs, or simply automating the process of generating inputs. In this thesis, we focus on one such automated process called BEACON or Branch-oriented Evolutionary Ant Colony OptimizatioN. We propose a modification to improve this method by using standard properties of the design. These properties, also known as invariants, help to cover those branches that require extra effort in terms of both inputs and time, and are thus, hard to cover. When we add these significant invariants to the design, modified BEACON is able to cover almost all accessible branches in the system with significantly less amount of time and lesser number of vectors than original BEACON itself, which helps save a lot of resources.

Dedication

Dedicated to my family.

Parents Naresh and Tuhina Khanna

Grandparents PremChand and Shanti Khanna and Usha Vohra

Brother Dhruv Khanna

Acknowledgments

Foremost, I owe my greatest thanks to my advisor, Dr. Michael S. Hsiao for granting me the opportunity to engage in his research. From the time of my applications to VT, I have been inspired by Dr. Hsiao's experience and contributions in the field and was lucky enough to have him as my interim advisor. His courses on "Electronic Design Automation" and "Testing and Verification" motivated me further to definitely pursue my Master's thesis under his guidance. Throughout my thesis, he guided and encouraged my work. I am honored to have work with him. I would like to thank Dr. Haibo Zeng and Dr. Lynn Abbott for agreeing to serve on my thesis committee. I would like to thank all my friends in the PROACTIVE Research Group who have supported me in my work- Kelson Gent, Akash Agarwal, Sonal Pinto, Tonmoy Roy and Aravind Krishnan. I would also like to thank Keepsake members- Shruti Modak and Aakanksha Sharma, Motu(Kunal Bansal), Shamit Bansal and Baba(Abhishek Bendre) for making my graduate school experience unforgettable. Lastly, I would like to thank my family and my constants- Aishwarya Chhabra, Priyanka Ashok and Avani Gupta, for their love and unwaivering support.

Contents

List of Figures ix

1 Introduction 1

1.1 Problem Scope . . . 1

1.2 Contributions of the Thesis . . . 3

1.2.1 Improvement of original BEACON algorithm using potential invariants 3

1.2.2 Double layer filter to obtain relevant potential invariants . . . 5

1.3 Thesis Organization . . . 5

1.4 Conferences . . . 6

2 Background, Instrumentation and Prerequisites 7

2.1 Automated Test Generation . . . 7

2.2 Related Work . . . 8

2.3 RTL Translation and Instrumentation . . . 11

2.4 Dynamic Analysis of Potential Invariants . . . 13

2.5 BEACON Algorithm . . . 15

2.5.1 Ant Colony Optimization . . . 16

3 Modification in BEACON Algorithm 19

3.1 Raw Invariants Types Reported 20
3.2 New and Modified Maps 21
3.3 False Invariant Search 22
3.4 Pheromone update 23
3.4.1 Reinforcement: 24
3.4.2 Evaporation: 25
3.5 Experimental Results 28
3.5.1 Algorithm Settings 29
3.5.2 Branch Coverage 30
3.5.3 Test Vector Quality 31
3.5.4 Runtime 31
3.5.5 False Invariant Coverage 32

4 Double-Layer Filter to Improve Potential Invariants List 33
4.1 Invariants Types Used 34
4.2 Filter Layer I 35
4.2.1 Invariants affiliated to certain variables: 35
4.2.2 Invariants affiliated to certain types: 36
4.3 Filter Layer II 36
4.4 Experimental Results 38

5 Clock Domain Crossing **42**

5.1 Background 42

5.1.1 Clock Domain 42

5.1.2 Metastability 43

5.2 Synchronization Techniques 43

5.2.1 Synchronization of CDC Control Signals 43

5.2.2 Synchronization of CDC Data Signals 44

5.3 CDC Analysis 45

5.3.1 Structural CDC Analysis 46

5.3.2 Functional CDC Analysis 47

6 Conclusion **51**

6.1 Conclusion Summary 51

6.2 Future Work 52

Bibliography **53**

List of Figures

1.1 Coverage over the entire verification cycle 3

2.1 (a) Verilog snippet and (b) its Verilator translated C++ RTL 12

2.2 Architecture of the Daikon tool for dynamic detection of program invariants[12] 13

2.3 (a) Daikon generated list of raw, unfiltered vs (b) list of non redundant likely invariants for a portion of b12 circuit 15

3.1 Preprocessing of BEACON Guidance Search with potential invariants injected 19

3.2 BEACON guidance framework with potential invariants injected 22

3.3 (a) Parsed C++ code for b12, (b) Reported invariants, (c) Invariants array injected in BEACON interface 27

4.1 Original invariants associated to b12 example 38

5.1 Double flip-flop synchronizer for Control Signal synchronization 47

5.2 Assertion for functional analysis of double flip-flop synchronizer[1] 48

5.3 MUX based synchronizer for Data Signal synchronization 49

5.4 Assertion for functional analysis of MUX based synchronizer[1] 49

Chapter 1

Introduction

With the increasing dependencies on the electronic devices in almost every possible activity, accuracy of designs has become the prime focus. To meet all advancing requirements of circuitry, designs are now reaching millions of gates and thus, the effort spent on verification of designs is only increasing exponentially with the sizes of the circuits. Repeated studies have shown that investment of significant resources and time in hardware design validation at the Register Transfer Level (RTL) are of utmost importance, both to avoid the high costs associated with design errors detected at later stages of development as well as to meet time-to-market window. Attention to structural metrics such as line coverage, block coverage, branch coverage and toggle coverage [7], can help report the sections of code exercised and later aid in affirming the correctness of the circuits.

1.1 Problem Scope

One of the earliest solutions involves simulation with random vectors and is reckoned as an unsuitable approach as it seldom achieves a reasonable coverage and traverses the same paths of execution recurrently. Random testing combined with directed testing certainly gives better coverage than random test vectors. The combination has proven to verify the functionality of designs much more efficiently; however, directed testing requires many hours of manual test generation. Complete and accurate test cases are not guaranteed even with all

the human effort. Even a considerably large set of test vectors may not be enough to ensure confidence in a design, as some areas of the design by specific sequences of test vectors. It is essential to automate the process to save as much manual labor as possible.

To do so, several algorithms have been proposed targeting RTL coverage as a crucial test metric. For this thesis, we concentrate on branch coverage as 100% branch coverage attests that every control state described in the RTL has been traversed and checked. A few semi-formal directed RTL test generators like HYBRO [25], PACOST [35] and [30] function on restraining the classical symbolic evaluation to analyze the real executable paths extracted during dynamic simulation using random vectors. Even though these hybrid algorithms achieve high coverage with a small set of test vectors, the computational cost for these techniques increases drastically with the size of the design due to the repeated calls to constraint solvers. Furthermore, targets that require very long sequences may be difficult to reach by such techniques. On the contrary, stochastic or heuristically guided search-based techniques [24][4] attain similar results relatively faster than formal and semi-formal techniques, but with a trade-off in form of comparatively larger set of test vectors.

Here, as a baseline test generator, we use BEACON [24], or Branch-oriented Evolutionary Ant Colony OptimizatioN, which is a scalable, simulation-based swarm intelligence technique that uses behavioral simulation of the Hardware Description Language (HDL) design and a self-refining Evolutionary Ant Colony Optimization (ACO) to generate the vectors. While BEACON gives good results, a number of hard-to reach corner cases of the design are either missed or require a large set of test vectors to achieve the same goals. Empirically, hard corner cases verification in a design constitute to only 18% of the total verification, but require approximately the same amount of time and resources as the other 80% broad spectrum-generic verification. Deducing from Fig.1.1, if the hard-to-reach cases are set as priority of the verification process and covered with minimal time and resources, then the

same resources can cover the remaining majority of the targets as a byproduct without extra efforts. Thus, we propose to use relevant potential invariants in the RTL circuits to generate improved, shorter vector sequences, without additional computational cost, to achieve the same or better branch coverage for a set of standard ITC99 benchmarks.

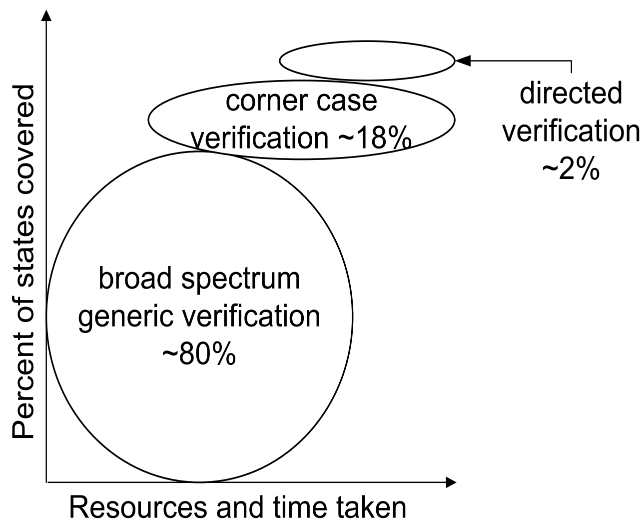

Figure 1.1: Coverage over the entire verification cycle

1.2 Contributions of the Thesis

The contributions of this thesis are as follows:

1.2.1 Improvement of original BEACON algorithm using potential invariants

A program invariant is a property relied upon to be true during execution of a program, or during some portion of it. There are different categories of invariants. For example, simple equations: $\mathbf{y = 4*x+3}$ and $\mathbf{x > abs(y)}$; constants: $\mathbf{x = 150}$; containment and conditions:

array A contains all elements greater than 8 and has no duplicates; linear relationship: **z = ax+by+c**; generic conditions: **graph G is planar**; **size(keys) = size(index)**. Since invariants are logical assertions that are true during a certain phase of execution, these prove to be beneficial, in all aspects of programming, especially testing, optimization, and maintenance. The invariant detection techniques can be static and dynamic analysis [13]. Static analysis examines the subject program for potential executions and runtime states. However, static analysis reportedly suffers from basic limitations such as high computational time and cost of modeling program states. Dynamic analysis on the other hand, reports properties of the subject program examined during every individual execution independently, in a fraction of the time and with low cost of modeling program states. For our methodology, we generate a basic list of potential invariants dynamically, using the Daikon.[13] The tool runs the program for a given set of stimuli, examines the executions and reports properties that hold true for the input stimuli, over those executions in a fraction of the time and with low cost of modeling program states. The derivation of these potential invariants indicate that these relations hold for the provided stimuli. However, there might exist stimuli that can prove them false. Hence, we generate additional stimuli in an attempt to violate these derived potential invariants. These generated stimuli may also help to reach previously uncovered branches, statements, etc. Our results show that the addition of these potential invariants as priority conditional statements in the existing BEACON algorithm helps to find many false potential invariants that were earlier missed by the original BEACON algorithm. This alteration also covers corner case states in tested ITC99 benchmarks, with considerable reduction in vector lengths to achieve same or in some cases, better branch coverage.

1.2.2 Double layer filter to obtain relevant potential invariants

The space of potential invariants can be very large and some could be irrelevant to the task of reaching hard-to-reach corners of the design. That is, Daikon may identify a very large set of potential invariants, many of which may contribute little to our goal of achieving high branch coverage. Therefore, we propose a two-layer filter to remove all the irrelevant invariants that only consume more computational time and extra test vectors for their falsification, but do not actually contribute in reaching any new or corner case branches. The first layer of filter is based on gathered results and observations. We foremost filter out invariants pertaining to certain variables and types that do not contribute to covering any new branches and only add to computational time. In the second layer, we simulate BEACON very quickly for just 1 cycle (typically just over a span of few microseconds) to detect the easy branches in the circuits. All the invariants affiliated to the variables of the already covered branches prove to be trivial. Thus, we flush out all those invariants in the second layer. Our results show that this two-layer filter presents much better results in terms of both lengths of vector sequences and runtime, with respect to the original BEACON, other test generation algorithms and unfiltered-raw list of potential invariants.

1.3 Thesis Organization

Rest of the thesis is as follows:

Chapter 2 describes gives an elaborate understanding of the background and prerequisites for the thesis. The chapter includes previous work algorithms on the same metric, all the tools used: Verilator [2] and Daikon and functioning of the original BEACON algorithm.

In chapter 3, we introduce our algorithm to add potential invariants as priority branches in

the existing BEACON algorithm and compare modified BEACON algorithm to the original BEACON and previous reported algorithms.

Chapter 4 elucidates our proposition of double layer filter to obtain only relevant potential invariants for the circuit under test, to save both computational time in checking redundant invariants and generation of extra test vectors as a buffer. We compare the results with original BEACON algorithm, modified BEACON with unfiltered invariants and previous reported algorithms.

Chapter 5 covers a theoretical understanding of systems with multiple asynchronous clock domains and the problems arising in such setups. We address a few issues and adopt a structural analysis ensure system correctness so that the invariants obtained for such designs are not void due to faulty system.

Chapter 6 gives the conclusion of our methodology.

1.4 Conferences

This work has been submitted in the following conferences for review:

- Tania Khanna and Michael Hsiao, "Using Potential Invariants to Guide Test Generation", International Test Conference 2018.
- Tania Khanna and Michael Hsiao, "Guiding RTL Test Generation Using Relevant Potential Invariants", 2018 IEEE International Conference on Computer Design.

Chapter 2

Background, Instrumentation and Prerequisites

In this chapter, we present the details of our base algorithm along with tools and fundamental concepts used in this thesis.

2.1 Automated Test Generation

Hardware validation allies to functional and structural correctness of the design, which is of prime importance in all devices. Structural metrics like line coverage, branch coverage, toggle coverage help monitoring structural and behavioral accuracy of the design. With growing complexity in circuit designs, exhaustive testing of even subsections of systems has become unachievable. Thus, there is need for automation of test generation to produce efficient test sequences, saving both time and resources to attain maximum structural and functional metrics and confirm the correctness at an earlier stage in the design cycle [31].

Of late, several automated test generations algorithms have been proposed. Directed search-based software testing techniques like [15], [32], and [26] adopt unrolling of the entire HDL every cycle of evaluation, and input set would grow unfeasibly with the length of the test. Known constraint-based formal technique of static symbolic execution [22] requires unrolling of the RTL design and evaluation of every statement per cycle. This technique is quite effi-

cient theoretically, as it makes every states in the RTL design reachable, but practically, it proves to be computationally inviable for real-world designs. Search-based techniques like BEACON [24] rely on heuristics measured over optimizing branch coverage among their candidate exploratory tests. Search-based techniques prove to be more popular due to their simplicity of implementation as they often require minimal instrumentation and can operate without severe knowledge of the underlying design, unlike constraint-based techniques. For the same reasons, we focus on search-based technique, BEACON for generation of high quality test vectors and reach maximum branch coverage possible. We also compare BEACON with other previous algorithms that use maximizing branch coverage as a base for test pattern generation.

We note that techniques such as BEACON can achieve high coverages, and the corresponding vector lengths generated could be high as well. Although methods to compact the vectors exist [29][20], they incur additional computational cost. Therefore, it would be beneficial to generate vectors that are short to start with.

2.2 Related Work

HYBRO [25] is a hybrid algorithm that uses static and dynamic program analysis techniques for test generation of Register Transfer Level (RTL) design source code. The algorithm uses concrete simulations for a fixed number of cycles and for every simulation, concrete traces are extracted from the design Control Flow Graph (CFG) [5]. The instrumented RTL code for the methodology maintains branch coverage counters per simulation cycle and any change in these counters qualifies as the execution path in the CFG. Starting from a fixed state and random input vectors, the extracted concrete traces are evaluated to identify covered branches (or guards) along the respective traces. The conditions and constraints for covered

branches are inverted and then fed as constraints through a Satisfiability Modulo Theory (SMT) solver, to generate new test vectors that explore the untraversed branches of the design. This approach proves to be inefficient because of its high computational cost of the SMT solver calls, especially for bigger circuits that require large vector sets of particular sequences to reach some branches. Moreover, HYBRO poorly relies on random stimuli to reach system states and does not support HDL array elements or memories, which makes it an unfit approach for practical designs.

Approach explained in [30] is quite similar to the HYBRO algorithm, with certain improvements. The algorithm uses dynamic analysis for concrete simulation and symbolic execution. The instrumented RTL, simulated with random stimuli, generates a trace file that has the syntactic objects printed after every simulation. A constraint solver, used to identify all the covered control statements symbolically, analyzes this output trace file. The unstimulated variables in all the covered control statements are substituted with their concrete equivalent. Since this algorithm uses dynamic program analysis, it can support symbolic evaluation of memory arrays by treating them as individual index-annotated scalar variables. However, like HYBRO, the process produces test vectors by simulating the design for a fixed number of cycles from a specific starting state and is limited to the exploration of system states reachable within the fixed number of simulation cycles.

PACOST [35], a path constraint solving-based test generator operates in an abstraction-guided semi-formal verification framework to cover hard-to-reach states. PACOST too uses concrete simulation and symbolic simulation of the design to obtain information on path constraints and mutations in the circuit, followed by a sequential path constraint extractor to generate valid input vectors to traverse different simulation paths beginning with different successor state. This algorithm is limited to obtaining information about path constraints for variables that can trace their use-definition chain to an input variable within that exploration.

If any variable in a covered branch path expression cannot be defined solely in terms of primary input, then no mutation effort is performed on that covered branch. In addition, abstraction of the RTL in PACOST leads to loss of information and the calculation of abstract state distances is limited by their Bounded Model Checking (BMC)-based preprocessing.

SMT solvers have been extensively used in HYBRO and PACOST. Each SMT instance symbolically encodes a specific execution path, which the SMT solver attempts to satisfy. If the constraint formula is satisfiable, it also generates the corresponding stimuli that exercises the corresponding execution path. One can also encode the instance as a model checking or bounded model checking instance, either with SMT or SAT formulation. Incremental solving by adding additional constraints have shown to improve the solver performance [34][6].

For this thesis, we use the original BEACON [24] algorithm, which has proven to be a more efficient approach as it is a search-based testing technique using a hybrid of Ant Colony Optimization and evolutionary algorithm to maximize branch coverage in the RTL design. In comparison to HYBRO, BEACON explores the circuit over a greater number of cycles and avoids the computational cost of SMT solver calls and still achieve a high level of coverage. Additionally, in contrast to PACOST, BEACON is guided by heuristic metrics derived from the overall coverage status of the test, and not just restricted to use-definition chains only confined within the exploration. Further on, we try to modify BEACON algorithm to generate shorter test vectors in shorter computational time, by guiding the algorithm with addition of invariants.

Program invariants, as known, are logical assertions that are true during a complete or certain phase of execution that prove to be beneficial in all aspects of programming especially testing, optimization, and maintenance. Dynamic invariants have been used for test generation as well, in particular for software verification [8]. The results of the reported framework prove that addition of extracted and verified properties as check point constraints to the

program can help improve the performance of software bounded model checking significantly through knowledge and search space reduction. Since the prime objective of hardware designs are growing analogous to software development, similar approach of adding invariants as constraints in our search based testing algorithm can improve test generation. Dynamically detected invariants can provide assistance to dispose unessential test sequences and infer when a test has failed.

Subsequent sections explain in detail about the tools used to generate instrumented HDL, likely invariants and theory of BEACON algorithm.

2.3 RTL Translation and Instrumentation

For fast simulation, a synthesizable high-level description is cross-compiled to a cycle-accurate behavioral model in a C++ base wrapper, using Verilator [2] tool.

Verilator is an open source Verilog HDL simulator that compiles synthesizable Verilog, some PSL, SystemVerilog and Synthesis assertions into a much faster optimized model, wrapped inside a C++/SystemC module. The translated behavioral model contains object oriented classes and functions. Foremost, *Vtop* function, which is structurally same as the top module in the Verilog HDL, contains all signals of the design declared as public data members. Moreover, the behavioral characteristic entirely models into a public member function called *eval*. Since this is a cycle-accurate behavioral model, stimuli can be provided by setting all primary-inputs, followed by two calls to *eval* with alternating values of **clock** input variable, one cycle to set the stimuli and the next to execute one complete cycle of the design.

In addition to this, the instrumented design is altered further during compile time to measure branch coverage. This alteration allows to track whether a block of code has been traversed,

at least once during simulation. Every block of code is defined by the entry and exit points in the form of control statements like case, if/else, and linked to counters that increment when the control flow executes through them. Statements with common entry and exit points are grouped as a single branch. For each branch the tool allocates a one-to-one mapping between branches in the compiled code and the original HDL (*VCoverage*). This *VCoverage* is a public C++ integer array in *Vtop*, maintaining the number of times each mapped branch is covered or hit during current simulation. Fig.2.1 depicts an example of a Verilog code and its Verilated conversion.

```
always @ (posedge clock )
  begin
      if (reset) //28
        timebase = 0;
      else //104
        begin
             if (start == 1'b1) //29
               gamma = `G1;
    else //30
               gamma = `G0;

             case (gamma)
               `G0: //31
                      begin
                           timebase = 0;
                      end

                    `G1: //32
                      begin
                           timebase = 33;
                      end
             endcase
        end
  end
```

(a)

```
if(reset) {
        ++(VCoverage[28]);
        timebase = 0;
}

else {
        ++(VCoverage[104]);
        if(start) {
                        ++(VCoverage[29]);
                        gamma = `G1;
        }
        else {
                ++(VCoverage[30]);
                gamma = `G0;
        }
        if(0==gamma) {
                ++(VCoverage[31]);
                timebase = 0;
        }
        else {
                ++(VCoverage[32]);
                timebase = 33;
        }
}
```

(b)

Figure 2.1: (a) Verilog snippet and (b) its Verilator translated C++ RTL

2.4 Dynamic Analysis of Potential Invariants

We use Daikon [3], which is an open source dynamic invariant detector that reports the possible invariants of a program at execution time by instrumenting the source program to trace the variables of interest. It executes the instrumented program over a test suite, and infers the likely invariants over both the instrumented variables and derived variables that are not evident in the program itself. Fig.2.2 shows the high-level architecture of the Daikon invariant detector.

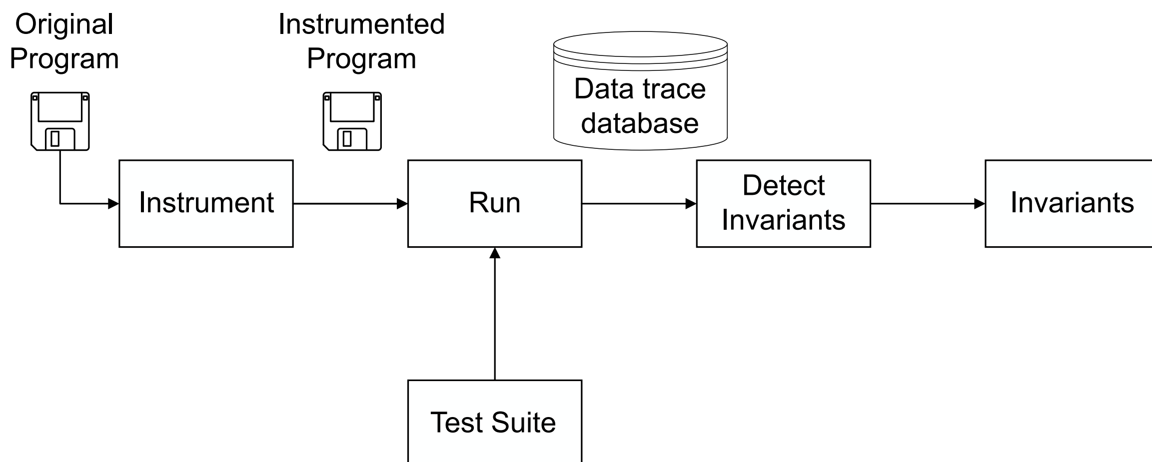

Figure 2.2: Architecture of the Daikon tool for dynamic detection of program invariants[12]

Since Daikon can detect potential invariants in C, C++, C#, Java, Perl, and Visual Basic programs, it is quite easy to extend this tool to other applications too. In addition, we have verified that this invariant detection method and tool constitutes to a very low one-time computational cost, which may be negligible in the overall cost, especially if the design is large with millions of gates and several thousand variables. Another advantage of using dynamically generated likely invariants is that the reported properties indicate exactly what inputs are needed to improve the test suite. For example, if the tool reported $\mathbf{a} > \mathbf{0}$ for certain set of test cases, it implies that the tool never simulated for values zero or negative

values for variable **a**. Similarly, for an outlined invariant $\mathbf{p} < \mathbf{20}$, no large values of variable **p** were exercised for the given test suite. Thus, these properties indicate exactly what inputs are needed to improve the test suite.

The invariants reported by Daikon depend on the grammar of invariants that is expressible by the invariant detector, the variables over which the invariants are checked, and the program points at which the invariants are checked [14]. The competency of these factors to report precise invariants is directly proportional to size of the test suites. Larger set of test cases can check and report more relations and properties with higher degree of correctness. To avoid an exhaustive check, there also exist techniques [17][33][16] that use a generate-and-check algorithm to test a set of potential invariants against the traces and finally produce good test suites for resultant invariants that are tested to a sufficient degree without falsification [27][28]. However, all these procedures come with a trade-off in the form of high computational time and cost for building these test suites, just like in case of static analysis.

For our method, Daikon is made to interact with the Verilator translated C++ to generate likely invariants for the original HDL designs. Since the *Vtop* function provides public access to all the variables of the HDL design, Daikon is run over random test suites to generate all the likely invariants of the circuit and later parsed such that it can be used for further analysis, which has been addressed in the ensuing sections. Since this execution provides stimuli to the circuit based on a random seed, not all invariants detected by the tool may be true invariants. Hence, the reported invariants are termed as "*likely*" or "*potential*" invariants. ITC99 b12 benchmark run in Daikon for random inputs like, **clock** always set to 1, gives a list of likely invariants for the given inputs. The same and list of the parsed, non redundant invariants are depicted in the below diagram.

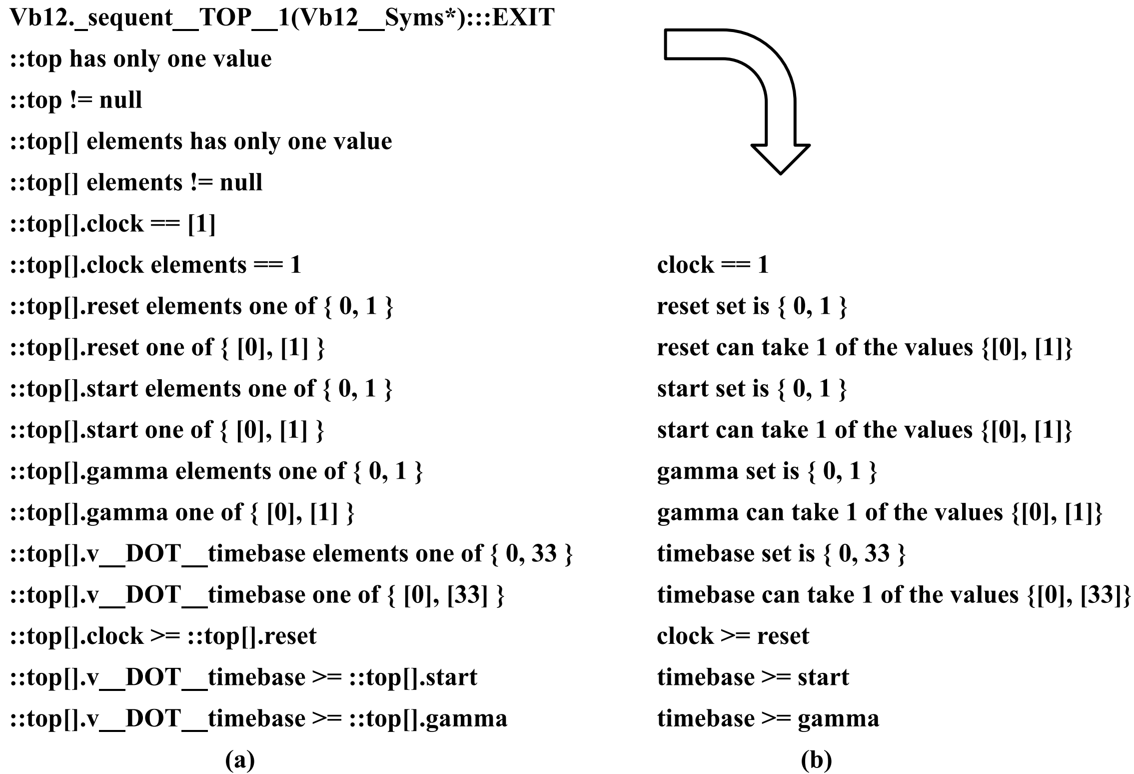

Figure 2.3: (a) Daikon generated list of raw, unfiltered vs (b) list of non redundant likely invariants for a portion of b12 circuit

2.5 BEACON Algorithm

The preprocessing of BEACON guidance search starts with the translation and instrumentation of HDL design, using Verilator, into a fast simulating behavioral model in C++ with direct access to all the signals and the branch points in the design. These branch points are stored in a database in an array, and later used to keep a track of the number of times the branches hit. BEACON directly accesses all these internal variables, states and the database of branches through an automatically generated interface. The control-flow graph of the code follows ACO algorithm [11][10], with minor alteration.

ACO is an evolutionary algorithm similar to genetic algorithms, except that the ACO utilizes swarm intelligence in addition to the behavior learned by the underlying population. Genetic

algorithms have been successfully used for test generation in [21][23][18][19].

2.5.1 Ant Colony Optimization

The ACO [11][10] is a probabilistic algorithm that models graph search as a scavenging simulation of an ant colony. The algorithm is based on swarm intelligence method, where multiple search units are modeled as artificial ants that communicate information using pheromone trails. This communication acts as the mechanism for reinforcement learning of the decentralized swarm and can be used as a meta heuristic for NP-hard search problems. Starting from an initial state, a population of ants begin a random walk through the edges of the search graph. Each ant in the colony deposits pheromones based on how favorable the transition is between two vertices in the graph. As ants pass these edges, they prefer paths with large amounts of pheromones, because these paths have been evaluated well by other ants within the colony and have a higher fitness.

Traditionally, ACO algorithm favors the trail with more pheromone deposits or branches that are easier to traverse. In our case, the edges in the design are the branches in the HDL code. However, the fitness function used in BEACON is a process opposite to the traditional ACO. High pheromone deposits in BEACON model make the trace less desirable, that is, it filters out highly visited branches from the search and focuses the search on rarely occurring branches and paths. This makes the ants more inclined to choose those branches that are not yet been covered.

The internal BEACON guidance framework initializes with:

- a pheromone map based on the database of branches with some constant initial value, say ϕ_b for each branch $\mathbf{b}$,

- a system with predefined set of ants, say **K**,
- a vector sequence of arbitrary cycles, say N_c,
- an upper bound for number of iterations or rounds, say **R**.

Once initialized, BEACON search proceeds by assigning randomly generated vector sequences of cycle N_c to each ant. All the **K** ants start out from the initial state S_0. Successively in each iteration, every ant conducts local search, where it uses its vector sequence to evaluate the behavior of the design and deposits certain amounts of pheromone on the traversed paths accordingly. After every round, the pheromone map is updated [24] and set of ants evolve, where each ant's succeeding vector sequences are computed according to a cut point determined by a fitness-proportionate selection procedure. All the states traversed during the simulation are stored in a pool of candidates, set_s and the cut points are selected from this pool. Mathematically, BEACON considers states with high fitness values to be hard-to reach, which in turn, help to discover new branches. Thus, selecting starting states within set_s is a stochastic process in terms of the fitness. This process of the stochastic selection is a function of the length of the vectors and fitness. This fitness is the sum of reciprocals of the pheromones deposited on branches within set B_c^k traversed in iteration **c** by ant ant_k [24].

$$fitness(S_c^k) = \sum_{b=0}^{B_c^k} \frac{1}{\phi_b} \quad (1)$$

Local search continues unless no new branches discovered for a predefined number of rounds, N_r. In such a case, it concludes that the number of cycles set as initial parameter is not sufficient to cover more branches. Thereafter, the starting state (S_0) updates from a set of states (set_s) which have been reached during previous local search with high fitness value. Once the starting state is determined, the set_s clears and local search executes again. The

whole process terminates when all the branches are covered or set_s is empty. If not the mentioned scenario, the search terminates after all **R** iterations.

Chapter 3

Modification in BEACON Algorithm

This chapter describes in detail the modifications made in original BEACON algorithm that help in reaching hard-to-cover corner states. The guidance in BEACON is modified by the addition of likely invariants generated by the Daikon tool, for any given design. Similar to the array of branches, these potential invariants are stowed in an array of integers and injected directly into the interface used by BEACON to interact with all the internal states and variables of the circuit. The high-level procedure of our approach, shown in Fig.3.1 and 3.2, describes the same two-step flow as the original BEACON.

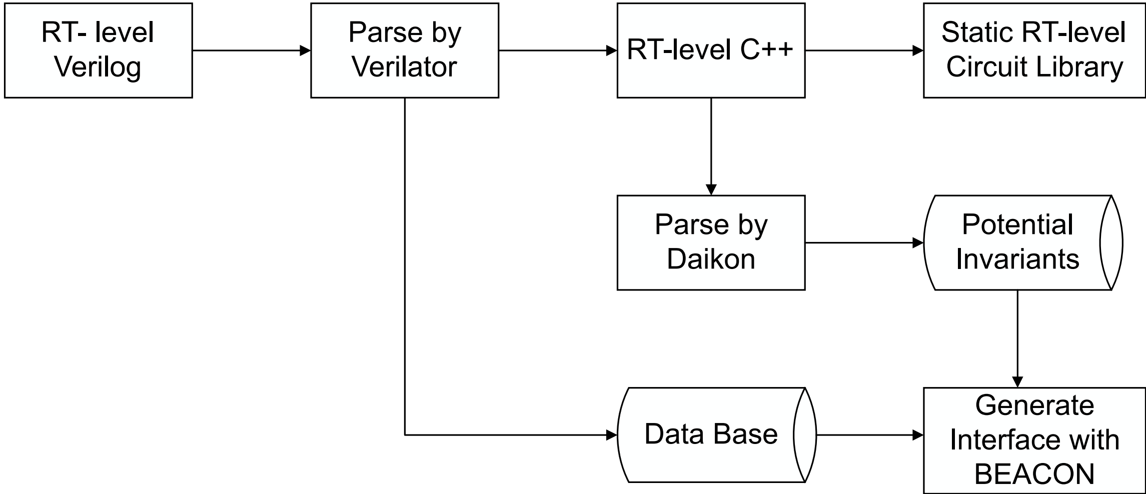

Figure 3.1: Preprocessing of BEACON Guidance Search with potential invariants injected

These inserted invariants and properties may or may not be true because are dynamically generated over a test suite with random stimuli and can serve as good checks for corner cases.

Hence, the number of false invariant among this set serve as a good guidance parameter to measure the diversity of input stimuli; that is, whenever a vector sequence is able to report a false invariant, that invariant array element is considered as traversed. For example, if ($\mathbf{a} < \mathbf{b}$) is a potential invariant reported, meaning that with random stimuli, the value of **a** is always less than the value of **b**, then if BEACON is able to generate a vector-sequence that violates this condition, then we have traversed a space that was initially difficult to reach by the random vectors.

3.1 Raw Invariants Types Reported

We first provide relatively large sized, random stimuli to the tool to generate a long list of invariants instantly and then use BEACON to get a list of good test suites to falsify invariants from the list without any compromise in computational time. Daikon is capable of checking and generating 75 different invariants types and allows users to easily extend and find more types. However, the output trace file generated by Daikon contains raw invariants along with several meaningless invariants. So, we simply proceed with our modification without such invariants types, for example,

- **Redundant invariants:** Random stimuli results in many invariants that are redundant and do not need to be checked. For instance, **clock = clock** or for any given design, if variable $a \leq array.length - 1$ then, $\mathbf{a} < \mathbf{array.length}$ are obvious and thus, categorized as redundant invariants. Since we use the translated and instrumented HDL design with the C++ wrapper as an input to Daikon, we get a large chunk of redundant invariants. Computing and checking for these redundant invariants only wastes resources and slows down the process. Therefore, we simply remove such invariants from our approach.

- **Abstract types:** For many programs under test, certain variables have no correlation with one another. For example, **angle** variable may store degree of rotation while variable **temp** may contain the integral value of the temperature in degrees. Even if there an invariant say, **temp** < **angle**, is reported by the tool, the two variables hold no meaning together to be actually related to one another. Daikon tool has settings to only restrict its search to related variables.

- **Incomparable variables:**Invariants types containing incomparable variables may be considered as a sub-category of abstract types. Not all variables can be sensibly compared. For instance, numerically comparison between a boolean and a non-null pointer, like **boolean** < **pointer value**, may be accurate but a useless invariant type. Similarly, comparing two integral values, one containing bits and other storing birthdate, makes no sense. It can be adapted as one of the user defined properties to restrict which variables should be compared by Daikon to increase the tool's performance further and reduce the number of irrelevant invariants reported.

We next discuss key modifications made to the existing BEACON algorithm.

3.2 New and Modified Maps

While all the other initial parameters in BEACON remain the same, the pheromone map is based on total of branches and invariants and initialized with initial constant value, say ϕ_b for each branch **b** and ϕ_i for each invariant **i**, where

$$\phi_i = \frac{\phi_b}{\chi} \tag{2}$$

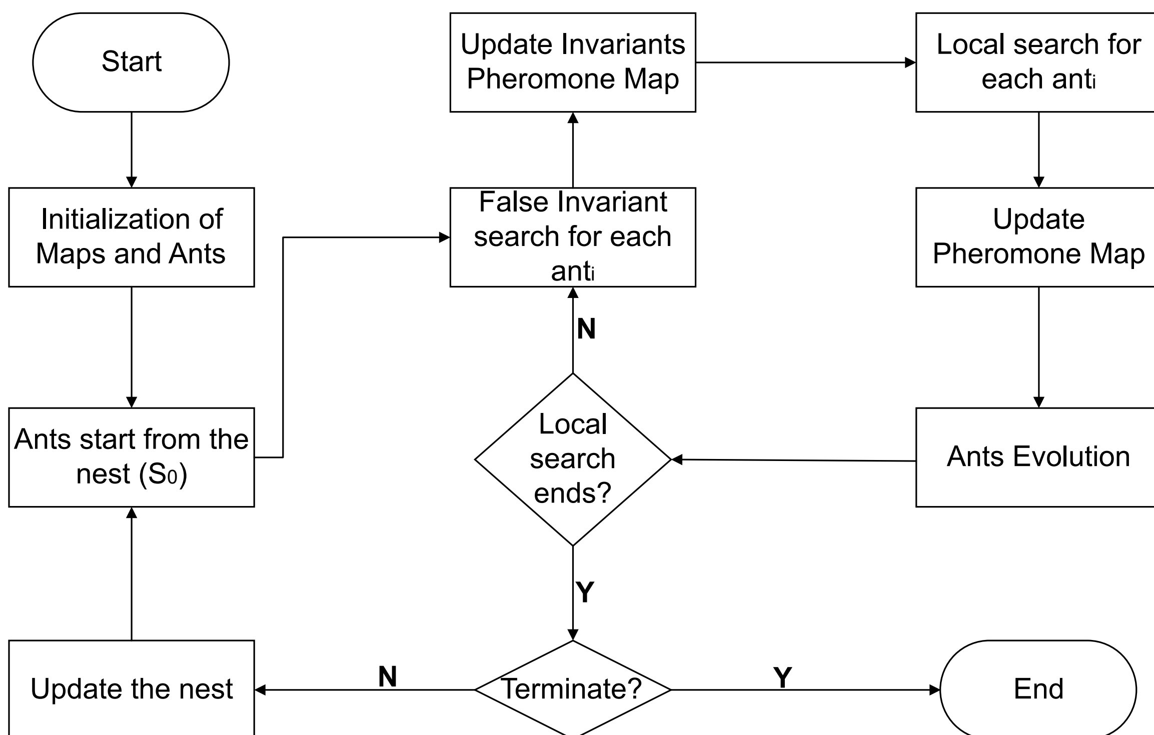

Figure 3.2: BEACON guidance framework with potential invariants injected

and **2**$\leq \chi \leq$ **number of rounds**, say **R**. In addition to the pheromone map, an invariants pheromone map is also maintained, that is based on only the invariants and initialized with $\phi_i$for each invariant **i**, as per equation (2). Like the original algorithm, all **K** ants start the search from the initial state S_0, with randomly generated vector sequences of cycle N_c.

3.3 False Invariant Search

Hereafter, in every iteration, each ant uses its vector sequence to conduct false invariant search process, where it only evaluates invariants and deposits equivalent pheromone on the paths of invariants that are detected as false, or as we count them as traversed. To achieve this, the *eval* function in Verilator generated C++ code is called to determine the true behavior and the false invariants reported for the design, for the input stimuli. After each

cycle, the invariants array updates for every traversed invariant. Like the original algorithm, here we maintain a set of states (set_s) reached and updated after every false invariant search with high fitness value. For the false invariant search process, if the simulation covers a new invariant or reaches a state with high fitness, the current state of the design is added to set_s as a candidate for starting states in next iteration. At this stage, only the invariants pheromone map is updated according to traversed traces. The detail of false invariant search is illustrated in Algorithm 1.

```
1  while rounds n < Nr do
2      for all ants k ← 1 to K do
3          for all cycles c ← 1 to Nc do
4              eval();
5              invariants array is updated;
6              if new branch || invariant traversed || fitness > threshold then
7                  sets ← state;
8              else
9              end
10             deposit pheromone on the path;
11         end
12     end
13     update pheromone();
14     if new branch is covered then
15         n = 0 //start again;
16     else
17         n++;
18     end
19 end
```

Algorithm 1: False Invariant Search

3.4 Pheromone update

After the false invariant search, the ants continue with their local search with the existing vector sequences, to assess the accurate behavior of the design, and deposit pheromones on

the paths that traverse both branches as well as invariants. After every round, the pheromone map is updated in two steps: reinforcement and evaporation. Since BEACON considers states with high fitness value to be hard-to reach, and fitness is inversely proportional to the pheromone variable ϕ as per equation (1), we have altered the pheromone updating procedures such that the conditions to prove invariants as false have higher fitness values and BEACON is forced to address these on priority.

3.4.1 Reinforcement:

The process of reinforcement deposits pheromones on the branches and invariants based on the number of times they are traversed. During the simulation for ant_k, the amount of pheromone deposited as a reinforcement: $\forall$ **b** $\in$ branches,

$$\phi_b(t+1) = \phi_b(t) + \frac{N_b}{N_c} * Q \tag{3}$$

where $\phi_b(t)$ is the amount of pheromone on branch **b** at time **t**. N_b is the number of times that the branch **b** is traversed in a total N_c cycles simulation [24]. **Q** is the maximum amount of pheromone released by one ant on one branches. This is the same as original BEACON algorithm. However, $\forall$ **i** $\in$ invariants,

$$\phi_i(t+1) = \phi_i(t) + \frac{N_i}{N_c * \chi} * Q \tag{4}$$

where $\mathbf{2} \leq \chi \leq \mathbf{R}$ and N_i is the number of times that the invariant **i** is traversed in a total N_c cycles simulation. The branches and invariants that are traversed more frequently will accumulate more pheromone. Higher the levels of pheromones, easier it is to cover the

branches and invariants; conversely, little or no pheromone indicate hard-to-reach states. We try to falsify as many invariants as possible to be able to generate better vector sequences in the subsequent iterations, to cover corner cases. Thus, we modify the pheromone variable for the invariants.

3.4.2 Evaporation:

The process of false invariant search and local search are together in a positive feedback loop executed by iterative adjustments of the simulated pheromone trails and ants evolution. In order to prevent this positive feedback loop from fixating on a suboptimal solution, pheromones disperse in the evaporation process. During the simulation for ant_k, the process of evaporation: $\forall$ **b** $\in$ branches,

$$\phi_b(t+1) = (1-\rho)+\phi_b(t) \tag{5}$$

where $\mathbf{0} < \rho \leq \mathbf{1}$ is the pheromone evaporation rate. This too is the same as original BEACON algorithm [24]. However, $\forall$ **i** $\in$ invariants,

$$\phi_i(t+1) = \frac{(1-\rho)}{\chi} + \phi_i(t) \tag{6}$$

The process allows access to all the hard to reach branches that are trimmed by the algorithm, if they are placed past an easy to reach branch. This provides a natural method for backtracking.

Thereafter, ants evolve using the same algorithm as earlier. The succeeding vector sequences are computed using the equivalent fitness-proportionate selection procedure that has the

same dependency on the length of the vectors and fitness. The fitness is also computed using equation (1), with only the pheromone variable ϕ changing as explained. Also, the subsequent processes of repetition and termination follow the same rules and conditions as the original BEACON.

To compare the original BEACON and our approach, we have used the same parsed C++ code for benchmark b12 as an example in Fig.3.3(a). This control-exhaustive circuit has a number of hard-to-cover branches. For simple illustration purpose, we skip signal declarations and use only a section of the design where the state variable, **gamma**, for the finite state machine (FSM) changes. The instrumented C++ code contains the *VCoverage* array to update the traversed branch points in the database. As per experiments, we note that branches *VCoverage[1]* and *VCoverage[6]* are hard-to-traverse corner cases and thus, have high fitness values. BEACON notes after multiple iterations that branch *VCoverage[6]* can only be traversed if all the bits of variable **k** are set to 0. Once this is traversed, the path acts as a guidepost to stay in state **G10**, decrement the variable count to 0 in multiple cycles and thus cover the branch *VCoverage[1]*. Original BEACON formulation requires the parameters to be set to: vector length $N_c > 100000$, the ant colony population $\mathbf{K} > \mathbf{200}$ and the maximum number of rounds $\mathbf{R} > \mathbf{50}$, to successfully traverse these two and many other corner case branches in benchmark b12.

Fig.3.3(b) depicts the invariants generated by Daikon, specific to the mentioned section of b12, based on random stimuli. It shows that the tool found that branches *VCoverage[1]* and *VCoverage[6]* cannot be traversed for random inputs. We add these invariants as new conditional branches in the BEACON interface, as shown in Fig.3.3(c), and increment the invariants array only if we can find inputs that falsify the properties stated in Fig.3.3(b). It is noteworthy that not all the properties and invariants reported by Daikon are false. In fact, for smaller designs, most of the invariants identified are true. As per our change in the

```
1.    if (gamma == G10) {
2.          VCoverage[0]++;
3.          if (count == 0) {
4.                VCoverage[1]++;
5.                gamma = K0;
6.          }
7.          else {
8.                VCoverage[2]++;
9.                count = count - 1;
10.               if (k[0] == KEY_ON) {
11.                     VCoverage[3]++;
12.                     if (data_out == 0) {
13.                           VCoverage[4]++;
14.                           gamma = G10a;
15.                     }
16.                     else {
17.                           VCoverage[5]++;
18.                           gamma = Ea;
19.                     }
20.               }
21.               .......
22.               else {
23.                     VCoverage[6]++;
24.                     gamma = G10;
25.               }
26.         }
27.   }
```

(a)

```
1.    gamma == G10 => count > 0
2.    gamma == G10 => k[*] > 0
```

(b)

```
1.    void clear_invariants() {
2.    for(int i=0;i<Invariants.size())
3.          Invariants[i] = 0;
4.    }
5.
6.    void check_invariants() {
7.    if(!(gamma == G10 && k[*] >0)) {
8.          ++Invariants[0];
9.          counter++;
10.         flag = 1;
11.         }
12.
13.   if(flag == 1 && counter < 32)
14.         //set inputs as same vector
15.
16.   if(!(gamma == G10 && count >0))
17.         ++Invariants[1];
18.   }
```

(c)

Figure 3.3: (a) Parsed C++ code for b12, (b) Reported invariants, (c) Invariants array injected in BEACON interface

algorithm of pheromone amount deposition, all the invariants array elements will have higher fitness than the original branches. Here, both *Invariants[0]* and *Invariants[1]* are on hard to reach path and thus will have very little accumulated pheromone, resulting in directing BEACON to cover these paths on priority. From the previous state of **gamma**, **count** was set to a value 32. As soon as BEACON covers *Invariants[0]*, the value of **count** decrements as per line 9 in Fig.3.3(a). We pass the same vector sequence to the circuit, till the value of count does not become 0. That helps BEACON to cover *Invariants[1]*. Experiments

have shown that BEACON takes only a few iterations to cover *Invariants[0]* and following the same path, it eventually covers *Invariants[1]* too. The results provide a set of vector sequences to prove that the reported invariants are false. At the same time, we are able to cover the hard-to-cover branches *VCoverage[1]* and *VCoverage[6]* in fewer iterations; and without any overhead computational time.

3.5 Experimental Results

The final translated HDL using Verilator and the input to Daikon both are in C++. Thus, all the intermediate variations and manipulations are in C++ too. The output of Daikon tool gives a long list of both useful and redundant invariants, in a raw format. We run a script to filter out the redundant invariants and parse this list in an opcode format such that, we can automate the process of adding invariant conditions in BEACON interface. All experiments run on a Linux (Lubuntu 15.10) machine with an Intel Core i7-975 (3.33GHz) with 6GB memory.

Since the quality and fitness of resultant potential invariants generated by Daikon depend on the size of test suites, the size of input stimuli to Daikon is set as big as the input parameters of BEACON. We use ITC99 [9] benchmarks to conduct our experiments, as many of these benchmarks are representatives of general sequential logic and have hard-to-reach states (control states), thus forming the perfect set to assess the performance of original BEACON and our implementation for efficient design validation. For most tested benchmark circuits, BEACON is set with an ant population $\mathbf{K = 50}$ and vector sequences of cycle $N_c = \mathbf{5000}$. Thus, the size of input stimuli to BEACON for the same ITC99 circuits is $\mathbf{K} * N_c \pm \mathbf{20\%}$. So the range of the size of the input set varies from 200,000 to 300,000 vectors. It is worth noting that even with such large input sets, Daikon generated list of invariants in fractions

of a second. Table3.1 depicts the number of raw invariants generated by Daikon and the number of non redundant invariants injected in BEACON to enhance its guidance.

Bench	raw, unfiltered invariants generated by Daikon	non redundant invariants added in BEACON
b06	1417	43
b07	1849	59
b10	2693	72
b11	2458	42
b12	3260	150
b13	5745	190
b14	6199	211
b15	6431	281

Table 3.1: COMPARISON OF NUMBER OF RAW INVARIANTS GENERATED BY DAIKON WITH MEANINGFUL INVARIANTS INSERTED IN BEACON

We analyze and compare the results, shown in Table3.2, based on various parameters: branch coverage, vector lengths for coverage, runtime to complete the execution, false invariants coverage.

3.5.1 Algorithm Settings

Foremost, experiments conducted showed in comparison to the original BEACON, our algorithm requires fewer resources to set up the parameters to initialize the BEACON algorithm. For the original BEACON, the stochastic ant colony search uses the following parameters: the ant colony population K=100; the maximum number of iterations or rounds R=15; vector sequences of arbitrary cycles Nc $\geq$ 3000. Compared to the mentioned data, our algorithm

cut down on the resources as much as K=20; R=10; Nc $\leq$ 3000 to give same or better results. These generic settings are not applicable to b12 circuit.

	# vectors		False Inv %		Branch Cov %		Time (s)	
Bench	Orig.	Mod.	Orig.	Mod.	Orig.	Mod.	Orig.	Mod.
b06	1731	**32**	0	**0.116**	**95.83**	**95.83**	**0.0054**	0.016
b07	759	**102**	27.12	**28.81**	**90**	**90**	0.37	**0.361**
b10	3547	**1033**	54.93	**64.79**	93.75	**96.87**	11.40	**8.46**
b11	**1235**	2088	2.38	**7.14**	96.88	**96.97**	11.95	**8.18**
b12	37006	**34339**	18.67	**44**	**98.09**	**98.09**	111.42	**37.08**
b13	5833	**4166**	33.68	**34.21**	92.19	**93.75**	**12.48**	13.07

Orig: Original BEACON Mod: Modified BEACON with injected potential invariants

Table 3.2: PERFORMANCE COMPARISON OF ORIGINAL BEACON AGAINST MODIFIED BEACON

3.5.2 Branch Coverage

Based on our experiments, original BEACON can cover very high branch coverage for most ITC99 benchmarks. With our approach, the branch coverage is as high as the original BEACON or in a few cases, higher coverage was achieved, such as in b10, b11, and b13. Furthermore, compared to the original algorithm, we retain a high coverage as well as attain significant improvements in test vector length without any compromise with the speed.

We also compare our results against HYBRO [25] and [30] in Table3.3. A quick glance at the results show that our work achieves a higher RTL branch coverage in significantly shorter test generation time. For example, in design b10 our algorithm achieves a high branch coverage score (**96.88%**) on par with HYBRO and [30] in only **8.46s**. Although HYBRO and [30] base their test generation effort on hybrid concrete and symbolic simulation, our work is more than **3**$\times$ faster for b10. For b11, our approach manages to cover all reachable branches, **33**$\times$ faster than previous methods.

	Branch Coverage %			Time (s) %		
Bench	HYBRO	[30]	Mod	HYBRO	[30]	Mod
b06	94.12	**96.30**	95.83	0.10	0.46	**0.016**
b10	96.77	96.67	**96.87**	52.14	24.61	**8.46**
b11	91.30	94.44	**96.97**	326.85	270.28	**8.18**

Mod: Modified BEACON

Table 3.3: PERFORMANCE COMPARISON WITH PREVIOUS ALGORITHMS

3.5.3 Test Vector Quality

Table3.2 depicts that our algorithm improves the length of vector sequences by extensive factors. For smaller designs like b06, the improvement is as much as **54** times since the addition of the invariants guide BEACON to prioritize the search toward them first. Consequently, the original branches were often covered as a side product. For the bigger circuits like b12 and b13, our approach trims most of the redundant vectors added to cover the hard corner states. As shown in Fig.1.1, when BEACON targets the branches corresponding to the invariants, it covers most or all of the corner cases and as a byproduct. This results in covering all the original branches with less number of vector sequences, in fewer iterations.

3.5.4 Runtime

Even though there is another tool used (Daikon), additional logics and checks are added in the BEACON interface (invariants array), the runtime of our algorithm is better than the original BEACON. This is because the hard corners are given higher priority and covered first without running multiple iterations with irrelevant vectors. With exception to b06 and b13, the run times for other circuits were reduced.

3.5.5 False Invariant Coverage

With random vectors, all the reported invariants are initially considered to be true invariants. Thus, the false invariant in that case is always 0. Even with original BEACON vectors, not all false invariants can be marked. With our algorithm, BEACON generates vector sequences that can filter false invariants and give better false invariant coverage for all the tested circuits.

Chapter 4

Double-Layer Filter to Improve Potential Invariants List

This chapter describes the modifications made to available potential invariants to filter out all the inconsequential invariants and provide a better guidance path for BEACON to thus, produce better test vector sequences for the designs and reduce the computational time.

The modified BEACON evaluates the entire hardware design twice, once to target only on the invariants introduced as conditional statements, and then again to focus on both invariants and the original branches of the circuit. For any potential invariant to be counted as traversed, the algorithm is directed to generate the vector sequences that prove the candidate to be false. In that process, BEACON mostly covers a large space of the circuit that was earlier hard to cover by the random test vectors, along with broad spectrum branches as a byproduct. As a result, all the original branches are covered with lesser number of vector sequences, in fewer iterations, as shown in Table4.2 and 4.3 respectively. While this verifies to improve BEACON in terms of branch coverage, vector sequences and relative computational time, the number of potential invariants injected in circuits would only increase exponentially with the size of the design. Thus, even if the candidate invariants prove to be a useful barometer to generate better vectors, the computational time to target these new conditional statements would be much more for bigger designs. Hence, it is of vital importance to get rid of all the invariants that do not have any significant contribution in

branch coverage.

4.1 Invariants Types Used

The stochastic ant colony search for BEACON used the following parameters: the ant colony population **K**; the maximum number of iterations or rounds **R**; and vector sequences of arbitrary cycles N_c. With given constraints, an appropriate stimuli for Daikon to generate invariants lists within a fraction of a second was estimated to be **K** * N_c ± **20%**, which roughly varied between 200,000 to 300,000 vectors. A few noteworthy and common invariants types used as a good fit for our approach to test and find corner case states are:

- **Constant values:** For example, $\mathbf{x = a}$; $\mathbf{x \neq 0}$.

- **Range definitions** $\mathbf{0 \leq x \leq 5}$ or $\mathbf{x} \in [\mathbf{0,5}]$, which is printed as **x** in [**0..5**] indicates the minimum and/or maximum value of x.

- **Sets:** All the values of a variable possible like, x ∈ $\{0, 1, 5\}$

- **Comparisons: clock ≤ setbit**

- **Equal variables:** For all variables that are equal, any invariant that is true for one of the variables will also be true for all the other variables as well. For example, if **a** = **b**, then for any invariant f, **f(a)** =>**f(b)**.

- **Linear relationships:** $\mathbf{y = ax + b}$; $\mathbf{z = ax + by + c}$.

- **Dynamically constant variables:** A dynamically constant variable has the same value at each observed sample. This type usually includes constants and equations.

For example, $\mathbf{x = 150}$ implies **x is even** and $\mathbf{x \geq 150}$. Similarly for equations, $x = 8$ and $\mathbf{y = 15}$ imply both $\mathbf{x < y}$ and $\mathbf{x = y\text{-}7}$.

- **Conditional invariants and implications:** These are the most important types of invariants so far. Variable relations used to define another invariant. The example in Fig.3.3(b) uses an implication invariant.

- **Variable Hierarchy:** There may exist variables whose values contribute to invariants conditions at multiple program points. For any two program points **x** and **y**, if all samples for **y** also exist at **x**, then any invariant true at **x** will be true and redundant at **y**.

We next discuss the filter layers to remove all immaterial invariants from the lists before adding to BEACON algorithm.

4.2 Filter Layer I

From the results with no filter on standard ITC99 benchmark circuits, it is noted that potential invariants affiliated to certain common variables and types do not play any role in enhancing branch coverage, but only add to computational time for the algorithm to try and search vector sequences to falsify these. Thus, in our filter layer I we get rid of these invariants such that modified BEACON only has to focus on the remaining filtered list.

4.2.1 Invariants affiliated to certain variables:

These variables are standard for all circuits. All invariants associated to variables **clock** and **reset** prove to be inconsequential, even if for certain cases they prove to be false invariants.

They do not define any properties that can contribute to any corner case or any new branch coverage.

4.2.2 Invariants affiliated to certain types:

Based on previous results and observations, it is noted that no matter how small or large the input stimuli to Daikon may be, the invariants type defining range of variables, like $\mathbf{x} \in \mathbf{(0,5)}$, are reported truly. Thus, it only consumes more time for the test generator to verify that these type of branches are in fact true for the tested circuits.

Thus, all these invariants types are removed from the set of potential invariants. Even though, the space of potential invariants reduces by a small ratio (as shown in Table4.1), there is a significant improvement in test vector sequence generated, as shown in Table4.2, with improved computational time, as depicted in Table4.3. BEACON no longer needs to generate buffer vectors, with extra time, to check these unimportant branches in the invariants list.

4.3 Filter Layer II

Filter layer I is based on previous records and observations. However, it is difficult to find a broad, generic classification of invariants that may prove to be insignificant in the guidance search, specially for larger circuits with as many as hundreds of variables. The invariant lists and types would only increase exponentially, and finding such a classification would only negate our original aim of minimizing the computational time for optimized vector set.

Thus, in our filter layer II, we move a step back and use the original BEACON algorithm on the circuit under validation. We set the initial parameters of BEACON to bare minimum

to complete the execution within microseconds: the ant colony population $\mathbf{K} = \mathbf{1}$; the maximum number of iterations or rounds $\mathbf{R} = \mathbf{1}$; and vector sequences of arbitrary cycles $100 \leq N_c \leq 2000$, depending on the size of the circuit and number of variables. With such settings, BEACON does not fully use ACO to guide the ant colony to complete the search and cover all branches possible. However, it quickly reports all the branches that can be easily fetched within 1 round and with a small set of random vectors. Using this as base, we start to filter out all the invariants containing variables included in these easily covered branches. The number of invariants to be injected in BEACON are reduced by a significant portion, as compared to the original list of non redundant invariants without any filters, as depicted in Table4.1. Interestingly, quite a few invariants filtered out in this layer were actually verified as false invariants by the modified BEACON. However, those only generate extra vector-sequences to falsify them and cover no new branch. Thus, in filter layer II, we remove all the invariants with variables associated to the branches that can be traversed without any ACO applied.

It is of utmost importance to note that neither of the two filters are applicable to any conditional or implication invariants. The example in Fig.4.1 depicts an extension of Fig.3.3(b). The relational invariant **gamma** $<$ **26** was present in the initial invariant list. Since gamma is the state variable for the FSM, a number of states were covered in a single round, with only 2000 random vector sequences and hence, we removed this relational invariant after filter layer II. However, the next two invariants are implication invariants that define the conditions for variables **count** and **k**, that were not associated to any branch covered in the initial stage. Thus, we save these invariants in the final invariants list. The same is true if **clock**, **reset** or range defining invariants are part of a conditional or implication invariant.

1. gamma < 26
2. gamma == G10 => count > 0
3. gamma == G10 => k[*] > 0

Figure 4.1: Original invariants associated to b12 example

4.4 Experimental Results

The size of input stimuli to Daikon for the same ITC99 circuits is in the same range from 200,000 to 300,000 vectors. With addition of the two layer filters, a significant number of invariants were removed, saving a lot of time in computing the overall branch coverage using modified BEACON. Table4.1 depicts the number of raw invariants generated by Daikon and the number of non-redundant invariants injected in BEACON without any filters and F1 and F2 as the two filter layer layers. It shows that layer I helps shed out only a few generic, observed types of invariants while layer II reduces the overall invariants list size by **50** to **70** **percent**.

Bench	raw, unfiltered invariants generated by Daikon	non redundant invariants added in BEACON	With Filter Layer I (F1)	With Filter Layer II (F2)
b06	1417	43	33	11
b07	1849	59	47	22
b10	2693	72	60	22
b11	2458	42	31	18
b12	3260	150	137	77
b13	5745	190	177	75

Table 4.1: NUMBER OF INVARIANTS AT DIFFERENT STAGES

Furthermore, we analyze and compare the results of our approach with the different stages of

	# vectors				Branch Cov %			
Bench	Orig.	No filter	Filter F1	Filter F2	Orig.	No filter	Filter F1	Filter F2
b06	1731	32	**22**	**22**	**95.83**	**95.83**	**95.83**	**95.83**
b07	759	102	65	**52**	**90.00**	**90.00**	**90.00**	**90.00**
b10	3547	1033	709	**362**	93.75	**96.87**	**96.87**	**96.87**
b11	1235	2088	931	**255**	96.88	**96.97**	**96.97**	**96.97**
b12	37006	34339	34327	**33816**	**98.09**	**98.09**	**98.09**	**98.09**
b13	5833	4166	3810	**3469**	92.19	**93.75**	**93.75**	**93.75**

Orig: no potential invariants used F1: Filter layer 1 F2: Filter layer 2

Table 4.2: COMPARISON OF VECTOR LENGTHS GENERATED BY ORIGINAL BEACON AGAINST OUR ALGORITHM WITH DOUBLE LAYER FILTER

modifications made in BEACON algorithm, as shown in Table4.2 and 4.3. Our results include various parameters: branch coverage, vector lengths, runtime to complete the execution. Based on earlier reported experiments, it is known that the original BEACON can already reach very high branch coverages for most ITC99 benchmarks. With our approach, the branch coverage is at least as high as the original BEACON or in a few cases, higher coverage was achieved, such as in b10, b11, and b13. In addition, compared to the original algorithm and modified BEACON with unfiltered invariants, we achieve shorter test vector lengths with reductions in execution time.

Table4.2 shows that our algorithm improves the length of vector sequences by extensive factors. For smaller designs like b06, the final vector count of **22** is approximately **79**× shorter than the original BEACON vectors of 1731, since the addition of the invariants guide BEACON to prioritize the search towards them first. Consequently, the original branches were often covered by these initial high-quality vectors as a side product. In order to reach the remaining branches, only a small number of vectors are needed in a few iterations. For the bigger circuits like b12 and b13, our approach trims most of the redundant vectors added

to cover the hard corner states. For example, for circuit b07, the original algorithm executes for multiple cycles and approximately 65% branches are covered in initial 2-3 cycles. And after every iteration, more vectors are added to the final list of vector sequences. With the two layer filter, the algorithm only focuses on the invariants and reaches as high as 80-85% in the first iteration itself. Thus, the number of cycles required to cover all the reachable branches also reduce. This is true for most circuits tested for our modification to the original algorithm.

	Branch Cov %				Time (s)			
Bench	Orig.	No filter	Filter F1	Filter F2	Orig.	No filter	Filter F1	Filter F2
b06	**95.83**	**95.83**	**95.83**	**95.83**	0.0054	0.016	0.0075	**0.0052**
b07	**90.00**	**90.00**	**90.00**	**90.00**	0.37	0.361	0.298	**0.027**
b10	93.75	**96.87**	**96.87**	**96.87**	11.40	8.46	6.77	**0.312**
b11	96.88	**96.97**	**96.97**	**96.97**	11.95	8.18	5.79	**0.353**
b12	**98.09**	**98.09**	**98.09**	**98.09**	111.42	37.08	36.83	**30.593**
b13	92.19	**93.75**	**93.75**	**93.75**	12.48	13.07	10.72	**8.943**

Orig: no potential invariants used F1: Filter layer 1 F2: Filter layer 2

Table 4.3: RUNTIME COMPARISON OF ORIGINAL BEACON AGAINST OUR ALGORITHM WITH DOUBLE LAYER FILTER

Even though there are Overheads: (1) another tool used (Daikon), (2) additional logics and checks are added in the BEACON algorithm (invariants array), (3) the original BEACON is run for 1 cycle for filter layer II, the runtime of our algorithm is faster than the original BEACON, as stated in Table4.3. This is because only the invariants associated to hard corners are included, given higher priority and covered first without running multiple iterations with irrelevant vectors. The speedup is as much as **37**× (for b10) and **34**× (for b11).

We also compare the results of our two layer filter against HYBRO and [30] in Table4.4. A quick glance at the results show that our work demonstrates significantly shorter test

	Branch Coverage %			Time (s) %		
Bench	HYBRO	[30]	Ours	HYBRO	[30]	Ours
b06	94.12	**96.30**	95.83	0.10	0.46	**0.0052**
b10	96.77	96.67	**96.87**	52.14	24.61	**0.312**
b11	91.30	94.44	**96.97**	326.85	270.28	**0.353**

Table 4.4: COMPARISON WITH PREVIOUS ALGORITHMS

generation time and achieves a higher RTL branch coverage. For example, in design b10 our algorithm achieves a high branch coverage score (**96.88%**) on par with HYBRO and [30] in only **0.312s**. Although HYBRO and [30] base their test generation effort on hybrid concrete and symbolic simulation, our work is more than at least **79**× faster for b10. For b11, our approach manages to cover all reachable branches, no less than **766**× faster than previous methods. We have reported results in Table4.4 only for the circuits reported for HYBRO and [30], except b14 circuit that we have not tested for our algorithm. The available b14 circuit code involved several VHDL operators, which are not completely compatible with the original Verilator tool that we have used and conversion of the code to complete Verilog was not accurate.

Chapter 5

Clock Domain Crossing

Most System-on-Chip designs are increasing in size and complexity with multiple asynchronous clock domains in the systems, thus requiring a clock domain crossing (CDC) based system. A CDC signal is a signal sampled from one clock domain to another asynchronous one or a clock domain with a variable phase relation and clasped by a flip-flop. Unsynchronized transferring of these CDC signals can lead to major glitches in the system. In this chapter, we address how the issues arising due to Clock Domain Crossing affect invariant generation of systems and some basic checks to avoid errors.

5.1 Background

5.1.1 Clock Domain

A clock domain is a part of a design that has a clock that functions asynchronous to, or has a variable phase relationship with another clock in the design. A single clock signal, its derived clock and its inverted clock are said to be synchronous because of same and constant phase relationship. However, three clock signals of frequencies 100MHz, 79MHz and 53 MHz would be asynchronous clock signals and thus, in different clock domains. From the given example, clock domain crossing signal would be a signal moving from 100MHz clock domain to 79MHz clock domain, or any of the other five permutation crossovers possible.

5.1.2 Metastability

All flip-flops in a system are designed with certain timing constraints namely, setup time and hold time, in which the data input should not be changed. Violation of either setup or hold time can cause the output of flop to be unpredictable for some time and the system reaches an unstable equilibrium or meta-stable state. Theoretically, the time taken by the system to settle to one state is not bounded and the output of a flop could even be oscillating several times before eventually settling down. As a result, the circuit can act in unpredictable ways, and may lead to glitches and a system failure.

In a design with multiple clock domains, meta-stability is inevitable because when a signal passes from one clock domain to another, the data can change at any time. To address CDC problems due to meta-stability, designers typically use synchronizers.

5.2 Synchronization Techniques

The output of flop in a meta-stable state oscillates between logics '0' and '1' for some time before settling to a constant value. In order to buy some extra time for the output to settle down to a stable value in the destination clock domain, designers add synchronizers in the design.

5.2.1 Synchronization of CDC Control Signals

- **Double flip-flop synchronizer:** The most common synchronizer used for CDC control signals is a double flip-flop synchronizer. In a double flop synchronizer, the first flip-flop samples the input signal from the source clock domain, say **Sclk** to the destination clock domain, say **Dclk**. Once the input is sampled on the **Dclk**, the system

waits for one complete **Dclk** cycle to allow any metastability after the first flop to decay. After one cycle is passed, the signal is sampled on same **Dclk** by the second flop of the synchronizer, with the hope that signal would now be stable and valid in **Dclk**.

Nonetheless, it is theoretically possible that the signal after first flop and one cycle wait is still unstable. Then the signal sampled by second flop could be meta-stable as well. However, metastability is only a probabilistic phenomenon and would converge to a stable value with time. For most synchronization applications, a double flop synchronizer is sufficient to remove all likely meta-stability.

5.2.2 Synchronization of CDC Data Signals

Since the data bus constitutes of more than one bit, ensuring accurate transfer of the entire data bus between clock domains is more challenging. Since any bit of the bus can change randomly between the clock domains, simple double flop synchronizer is not an ideal solution. There are three known methods used for synchronizing data between clock domains are:

- **Using MUX based synchronizer**: In a MUX based synchronizer, the control signal is first synchronized using a double flop synchronizer and then the synced control signal is passed as select line, while the data signal is passed as input to the MUX to synchronize the data bus in destination clock domain.

- **Using Handshake signals**: In this technique, a set of handshake control signals are introduced in the system. The data is placed on a data bus between the source clock domain, say **Sclk** and destination clock domain, say **Dclk**. The **Sclk** domain first sends a "**request**" signal to **Dclk** domain through a synchronizer. When the

Dclk domain receives the **request** signal, it clocks the data from the data bus into a register and after the data is sampled stable for two or three clock cycles, sends an "**acknowledgement**" signal to **Sclk** domain through another synchronizer. Once **Sclk** receives the **acknowledgement** signal, only then the value of the data bus is changed. Until then the data on the bus remains constant. This protocol has several disadvantages. Foremost, the latency to pass data from one clock domain to another increases with every additional control signal. In addition to this, the latency required to pass and recognize all of the handshaking signals for each data word that is transferred adds to the total time as well.

- **Using "First In First Out" (FIFO) shared memory to write data in one clock domain and to read data in another clock domain**: This protocol is one of the most used methods. A dual port shared memory is used for the FIFO storage. One port is controlled by the **Sclk** domain to write data in memory at the speed of one data word per write clock. The other port is controlled by the **Dclk** domain to read data out of memory at the rate of one data word per read clock. Two separate control signals indicate whether FIFO is empty, full, or partially full while two more control signals flag if the FIFO is almost full or almost empty.

Other than these known synchronizers, there can be user defined synchronizers as well. In addition to using synchronizers, CDC analysis needs to be done to verify functionality of the design and accuracy of clock domain crossings.

5.3 CDC Analysis

There are two broad classifications of CDC analysis:

5.3.1 Structural CDC Analysis

For this thesis, this is the main focus and most work and research for this was done during the 6 months internship at Intel Corporation. In the process of fetching invariants in the design, structural accuracy of the circuit is very crucial. In case of multi clock domain systems, it is mandatory to add synchronizers. However, there could be cases where synchronization in the setup is not adequate. Structural clock domain crossing analysis looks for issues leading to insufficient or incomplete synchronization of the system.

Some of these issues could be because of:

- Faulty combinational logic before the flip-flop based synchronizers.
- Multi-bits driving flops of synchronizers.
- Fan-in from different flops, which is similar to multi-bits. Here the input could be driven from multiple flops.
- Missing double flop synchronizers in the system.

For the thesis purpose, we used Questa Sim tool by Mentor Graphics to conduct structural analysis. All the violations were thoroughly studied and fixed for a model type display IP of Intel Corporation. The tool reports all the structural CDC violations, along with clock, rest, and unit details. Since this analysis is to attain sufficient synchronization through synchronizers, the violations are fixed by adding and fixing synchronizers and logic in the main source code.

5.3.2 Functional CDC Analysis

Functional CDC analysis is assertion-based verification to check the correct usage of synchronizers. Even if synchronizers eliminate the issue of metastability, improper synchronization may also result in functional CDC errors. For instance, there could be data stability violations when going from a fast clock domain to a slower one. Just having a synchronizer connected is only half the solution; the associated logic must interact correctly with the remaining circuit to ensure functional correctness. To ensure this, assertions are used to validate correct use of the synchronizers and check the correct functionality of the system as a whole. Some of the assertions added for the synchronizers are as follows:

- **For double flip-flop synchronizer**: Since double flop synchronizers are only used to synchronize the control signal, assertions are added to verify the control signal stability after the synchronizer.

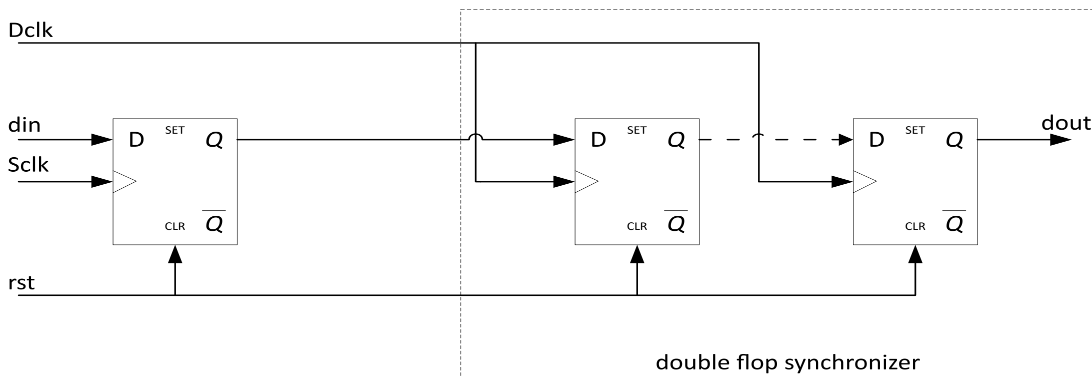

Figure 5.1: Double flip-flop synchronizer for Control Signal synchronization

A basic double flop synchronizer (Fig.5.1) consists of 2 flops modeled as a shift register. Flip-flop synchronizers can have more flops added in series for synchronization but 2 flops usually suffice. The purpose of this synchronizer is to sample the input data

(**din**) from source clock domain, say **Sclk** into destination clock domain , say **Dclk** accurately as **dout**. To confirm correct functionality, assertion is added to check that if the synchronizer has 2 flops, the input data should be stable for at least 3 **Dclk** cycles.

The assertion for the same in System Verilog in shown below:

```
property control_signal_check;
disable iff (rst)
@(posedge Dclk)
!$stable (din) |=> $stable
(din)[*2] );
endproperty

Stability_check: assert property (control_signal_check);
```

Figure 5.2: Assertion for functional analysis of double flip-flop synchronizer[1]

Similarly for all the other mentioned synchronizers, assertions are added to make sure that the functionality of all the synchronizers is right.

- **For MUX based synchronizer**: For a given MUX based synchronizer (as shown in Fig.5.3), first the control signal is synchronized using a double flop synchronizer and then the synced-in control signal is used to synchronize data signal. In addition to control signal stability assertions for synced-in select line of the MUX, data signal stability assertions are added too.

  To confirm the correct working of the synchronizer, we assert that the control signal "**select**" is synchronized for at least 3 **Dclk** cycles. Also, the input data should remain stable in **Dclk** domain while data is sampled from **Sclk** to **Dclk**. The assertion will be as follows:

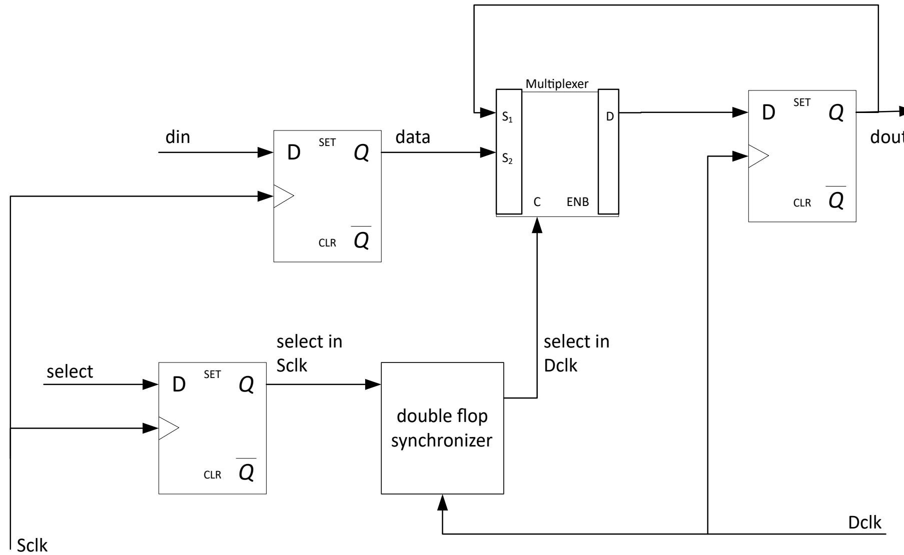

Figure 5.3: MUX based synchronizer for Data Signal synchronization

```
property data_signal_check;
@(posedge Dclk)
((select_in_Dclk) |=> ($stable (data) || (!select_in_Dclk));
endproperty

Stability_check: assert property (data_signal_check);
```

Figure 5.4: Assertion for functional analysis of MUX based synchronizer[1]

- **For handshake synchronizer**: The data signal stability assertions for handshake synchronizer follows the same protocol as MUX based synchronizer. However, there are two control signals in handshake synchronizer, one for **request** and other for **acknowledge**. With the same logic followed for double flop synchronizer, assertions are added till request is stable for at least 3 destination clock cycles and same for acknowledge signal in source clock cycles.

- **For FIFO based synchronizer**: FIFO based synchronizers include multiple assertions to check:

  1. Data integrity, such that FIFO preserves the right order and data value.

2. Gray coded at source for read and write pointers.
3. Control signal stability.
4. FIFO protocol is followed correctly so that user does not write when FIFO is full and read when FIFO is empty.

Chapter 6

Conclusion

6.1 Conclusion Summary

In this thesis, we propose a modification to the existing BEACON algorithm to achieve equal or higher branch coverage along with considerable reduction in vector lengths and computational cost. To do so, we focus on using relevant potential invariants of the design in the existing algorithm and direct BEACON such that it addresses the added invariant conditions as priority cases and cover those paths first. Our approach helps to cover many of the hard-to-traverse states. Further, the initial list of invariants consists of a number of properties that only use resources to either confirm their correctness or even if they are wrong, do not contribute in covering any new or hard-to-cover branches. Thus, the list of meaningful invariants is fetched by using a two layer filter to get better results by saving resources. Experiments conducted on several ITC99 benchmarks have shown considerable improvement in performance for our methodology over original BEACON, including equal or higher coverages, shorter test lengths, and reduced execution time.

After the introduction and background chapters, Chapter 3 focuses on a variation in the original BEACON algorithm. Instead of focusing on the original branch points of the circuit, we direct the algorithm to use invariants as control statements and treat those as priority branches, by adjusting the pheromone deposit on these additional, new statements paths accordingly. The experimental results show same or better branch coverage in most cases,

along with a considerable improvement in the length of test vectors thus, confirming that if we focus on hard-to-cover corner cases, then majority cases would be covered as a by-product, without buffer vectors. It may also be noted that the computational time is better in most cases, but the improvement is not by large magnitudes.

Chapter 4 addresses this problem and helps explain solution to generate smaller sets of test vectors in relatively lesser computational time. For most circuits tested, the list of invariants contain properties that consume time and resources to confirm their correctness; whether true or false, but do not have any contribution towards the main aim. With doubler filter, all the insignificant invariants are removed from the list of priority conditions. The experimental results show a noteworthy improvement in both vector lengths and computational time.

Chapter 5 covers a theoretical understanding, which can be explored in the future for larger designs with multiple clock domains. It is essential to address instability and errors that can occur in a design due to clock domain crossing.

6.2 Future Work

As mentioned earlier, Daikon can identify as many as 75 types of invariants. In addition to that, the tool can be altered at user end to generate particular types of invariants. The scope of this research was only limited to a few commonly reported invariants types. For future work, we can explore some other settings of the tool for larger systems. The current default invariants types are sufficient for circuits we tested, but may not be good enough for bigger designs. A better set of invariants types would allow better guidance to BEACON and would also allow us to test and classify more layers of filters, if possible.

Bibliography

[1] Clock domain crossing. URL https://filebox.ece.vt.edu/~athanas/4514/ledadoc/html/pol_cdc.html.

[2] Verilator, . URL http://www.veripool.org/wiki/verilator.

[3] Daikon, . URL https://plse.cs.washington.edu/daikon/download/doc/daikon.html#Kvasir.

[4] V. V. Acharya, S. Bagri, and M. S. Hsiao. Branch guided functional test generation at the rtl. In *2015 20th IEEE European Test Symposium (ETS)*, pages 1–6, May 2015. doi: 10.1109/ETS.2015.7138737.

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

[6] R. Arora and M. S. Hsiao. Enhancing sat-based bounded model checking using sequential logic implications. In *17th International Conference on VLSI Design. Proceedings.*, pages 784–787, 2004. doi: 10.1109/ICVD.2004.1261028.

[7] Boris Beizer. *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990. ISBN 0-442-20672-0.

[8] X. Cheng and M. S. Hsiao. Simulation-directed invariant mining for software verification. In *2008 Design, Automation and Test in Europe*, pages 682–687, March 2008. doi: 10.1109/DATE.2008.4484757.

[9] S. Davidson. Itc'99 benchmark circuits - preliminary results. In *International Test Conference 1999. Proceedings (IEEE Cat. No.99CH37034)*, pages 1125–1125, 1999. doi: 10.1109/TEST.1999.805857.

[10] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, Nov 2006. ISSN 1556-603X. doi: 10.1109/MCI.2006.329691.

[11] Marco Dorigo. Optimization, learning and natural algorithms. *PhD Thesis, Politecnico di Milano*, 1992.

[12] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb 2001. ISSN 0098-5589. doi: 10.1109/32.908957.

[13] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants.* Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[14] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, December 2007. ISSN 0167-6423. doi: 10.1016/j.scico.2007.01.015. URL http://dx.doi.org/10.1016/j.scico.2007.01.015.

[15] R. Ferguson and B. Korel. Software test data generation using the chaining approach. In *Proceedings of 1995 IEEE International Test Conference (ITC)*, pages 703–709, Oct 1995. doi: 10.1109/TEST.1995.529900.

[16] N. Gupta and Z. V. Heidepriem. A new structural coverage criterion for dynamic

detection of program invariants. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 49–58, Oct 2003. doi: 10.1109/ASE.2003.1240294.

[17] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 60–71, May 2003. doi: 10.1109/ICSE.2003.1201188.

[18] M. S. Hsiao, E. M. Rudnick, and J. H. Patel. Sequential circuit test generation using dynamic state traversal. In *Proceedings European Design and Test Conference. ED TC 97*, pages 22–28, Mar 1997. doi: 10.1109/EDTC.1997.582325.

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

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

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

[22] A. Kolbi, J. Kukula, and R. Damiano. Symbolic rtl simulation. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 47–52, 2001.

[23] D. Krishnaswamy, M. S. Hsiao, V. Saxena, E. M. Rudnick, J. H. Patel, and P. Banerjee.

Parallel genetic algorithms for simulation-based sequential circuit test generation. In *Proceedings Tenth International Conference on VLSI Design*, pages 475–481, Jan 1997. doi: 10.1109/ICVD.1997.568180.

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

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

[26] Phil McMinn and Mike Holcombe. Evolutionary testing of state-based programs. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 1013–1020, New York, NY, USA, 2005. ACM. ISBN 1-59593-010-8. doi: 10.1145/1068009.1068182. URL http://doi.acm.org/10.1145/1068009.1068182.

[27] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 229–239, New York, NY, USA, 2002. ACM. ISBN 1-58113-562-9. doi: 10.1145/566172.566213. URL http://doi.acm.org/10.1145/566172.566213.

[28] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. *SIGSOFT Softw. Eng. Notes*, 27(6):11–20, November 2002. ISSN 0163-5948. doi: 10.1145/605466.605469. URL http://doi.acm.org/10.1145/605466.605469.

[29] M. S. Nsiao, E. M. Rudnick, and J. H. Patel. Fast algorithms for static compaction of sequential circuit test vectors. In *Proceedings. 15th IEEE VLSI Test Symposium (Cat. No.97TB100125)*, pages 188–195, Apr 1997. doi: 10.1109/VTEST.1997.600260.

[30] X. Qin and P. Mishra. Scalable test generation by interleaving concrete and symbolic execution. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, pages 104–109, Jan 2014. doi: 10.1109/VLSID.2014.25.

[31] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM. ISBN 1-59593-014-0. doi: 10.1145/1081706.1081750. URL http://doi.acm.org/10.1145/1081706.1081750.

[32] Joachim Wegener, Andre Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841 – 854, 2001. ISSN 0950-5849. doi: https://doi.org/10.1016/S0950-5849(01)00190-2. URL http://www.sciencedirect.com/science/article/pii/S0950584901001902.

[33] Tao Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 40–48, Oct 2003.

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

[35] Y. Zhou, T. Wang, H. Li, T. Lv, and X. Li. Functional test generation for hard-to-reach states using path constraint solving. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(6):999–1011, June 2016. ISSN 0278-0070. doi: 10.1109/TCAD.2015.2481863.