Title
Applications of Formal And Semi-formal Verification on Software Testing, High-level Synthesis And Energy Internet

Author
Gao, Min

Publication Date
2018

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
Los Angeles

Applications of Formal And Semi-formal Verification on Software Testing, High-level Synthesis And Energy Internet

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Electrical And Computer Engineering

by

Min Gao

2018

ABSTRACT OF THE DISSERTATION

Applications of Formal And Semi-formal Verification on
Software Testing, High-level Synthesis And Energy Internet

by

Min Gao
Doctor of Philosophy in Electrical And Computer Engineering
University of California, Los Angeles, 2018
Professor Lei He, Chair

With the increasing power of computers and advances in constraint solving technologies, formal and semi-formal verification have received great attentions on many applications. Formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property. These verification techniques have wide range of applications in real life. This dissertation describes the applications of formal and semi-formal verification in four parts. The first part of the dissertation focuses on software testing. For software testing, symbolic/concolic testing reasons about data symbolically but enumerates program paths. The existing concolic technique enumerates paths sequentially, leading to poor branch coverage in limited time. We improve concolic testing by bounded model checking. During concolic testing, we identify program regions that can be encoded by BMC on-the-fly so that program paths within these regions are checked simultaneously. We have implemented the new algorithm on top of KLEE and called the new tool Llsplat. We have compared Llsplat with KLEE using 10 programs from the Windows NT Drivers Simplified and 88 programs from the GNU Coreutils benchmark sets. With 3600 second testing time for each program, Llsplat provides on average 13% relative branch coverage improvement on all 10 programs in the Windows drivers set, and on average 16% relative branch coverage improvement on 80 out of 88 programs in the GNU Coreutils set.

The second part of the dissertation implements symbolic/concolic testing methods onto an embedded platform. With the more extensive use and of higher demand of the embedded systems, reliability of the embedded software becomes a critical issue. Thus it is important to design a test harness that can test embedded software on the real platform or hardware in the loop framework comprehensively and systematically. We present our design prototype Codecomb. Codecomb implements symbolic/concolic execution that is able to achieve high branch coverage to generated test cases. It mainly exploits client/server architecture to achieve the isolation of testing tools and program under test such that complex computing job is performed in the server side. Experimental results show that Codecomb can detect program deficiency automatically on the embedded platform, and precisely locate errors such as buffer overflow, memory leak in a running program.

The third part of the dissertation applies formal and semi-methods to high-level synthesis (HLS) for VLSI. Verifying functional equivalence of high-level synthesis with formal methods ensures the correctness of the transformation flow. Current verification work widely uses static analysis such as model checking, while a pure dynamic execution flow is missing. In this part, we propose a functional verification flow for HLS utilizing symbolic execution on both C and Verilog directly. Specifically, on behavior C level we collect program traces via symbolic execution. As for Verilog level, we first generate a circuit satisfiability modulo theory (SMT) representation. Then we propose a light-weight pure symbolic execution framework to collect Verilog's on-the-fly time invariant version-based traces. To alleviate the scalability issue, we develop an operation abstraction method using SMT solvers to match potential C and Verilog traces. Extensive experiments on circuits from numerical computing and Chstone benchmark verify the validity and effectiveness of the flow.

The last part of the dissertation investigates the applications on Energy Internet. Energy Router based system is a crucial part in the energy transmission and management under the circumstance of Energy Internet for green cities. During its design process, a sound formal verification and a performance monitoring scheme are needed to check its reliability and meaningful quantitative properties. In this chapter, we provide formal verification solutions for ER based system by proposing a continuous-time Markov chain model describing the

architecture of ER based system. To verify real world function of the ER based system, we choose electricity trading to propose a Markov decision process model running on an ER subsystem to describe the trading behaviour. To monitor the system performance, we project the energy scheduling process in ER based system, and then implemented this scheduling process on top of cloud computing experiment tool. Finally, we perform extensive experiment evaluations to investigate the system reliability properties, quantitative properties, and scheduling behaviours. The experiment verifies the effectiveness of the proposed models and the monitoring scheme.

The dissertation of Min Gao is approved.

Todd D Millstein

Sudhakar Pamarti

Puneet Gupta

Lei He, Committee Chair

University of California, Los Angeles

2018

To my family

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Firstly, I would like to express my sincere gratitude to my advisor Prof. Lei He for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Puneet Gupta, Prof. Sudhakar Pamarti, and Prof. Todd Millstein, for their insightful comments and encouragement, but also for the hard question which help me to widen my research from various perspectives.

My sincere thanks also goes to Prof. Rupak Majumdar, Dr. Gary Yeap, Dr. Richard Ho and Prof. Chang Wu, who provided me opportunities to join their team as intern, and who gave access to the laboratory and research facilities. Without they precious support it would not be possible to conduct this research.

I thank my fellow mates in for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had during these years. In particular, I am really grateful to Dr. Zilong Wang for enlightening me the first glance of research.

Last but not the least, I would like to thank my family: my parents and my grandparents for supporting me spiritually throughout writing this thesis and my life in general.

VITA

2011 B.E. (Electrical Engineering and Automation) and B.A. (English), Nanjing University of Technology, Nanjing, China.

2012 M.S. (Electrical Engineering), UCLA, Los Angeles, California.

2014 Visiting Scholar, Max Planck Institute for Software Systems, Kaiserslautern, Germany.

2015 Visiting Scholar, Max Planck Institute for Software Systems, Kaiserslautern, Germany.

2016 PhD Intern, Synopsys, Sunnyvale, California.

2017 Visiting Scholar, Shanghai Fudan Microelectronics Group, Shanghai, China.

2017 PhD Intern, Google, Sunnyvale, California.

2013–Present Teaching Fellow/Research Assistant, Electrical And Computer Engineering Department, UCLA, Los Angeles, California.

PUBLICATIONS

Ayca Balkan, **Min Gao**, Paulo Tabuada, Lei He,"A Behavioral Algorithm for State of Charge Estimation," the 26th Electric Driving Transportation Association Electric Vehicle Symposium (EVS'12).

Qiang Li, Kun Wang, Suwei Wei, Xuefeng Han, Lili Xu, **Min Gao**,"A Data Placement Strategy based on Clustering and Consistent Hashing Algorithm in Cloud Computing," 9th International Conference on Communications and Networking in China.

Yuhua Zhang, Kun Wang, **Min Gao**, Zhiyou Ouyang, and Siguang Chen, "LKM: A LDA-Based K-Means Clustering Algorithm for Data Analysis of Intrusion Detection in Mobile Sensor Networks," International Journal of Distributed Sensor Networks, Article ID 491910.

Kun Wang, Yuhua Zhang, Lei Shu, Chunsheng Zhu and **Min Gao**, "NAPR: A Node Activity-based Probabilistic Routing Algorithm in Delay Tolerant-Mobile Sensor Networks," IEEE International Conference on Communications 2015.

Wensheng Guo, Guowu Yang, Xiaoyu Li and **Min Gao**, "A SAT-based Algorithm for Finding Cycles in a Boolean Network," Journal of University of Electronic Science and Technology of China, vol. 44 issue 6, Dec 2015.

Hui Jiang, Kun Wang, Yihui Wang, **Min Gao** and Yan Zhang, "Energy Big Dta: A survey" IEEE Access, vol 4.

Min Gao, Lei He, Rupak Majumdar, Zilong Wang, "LLSPLAT: Improving Concolic Testing by Bounded Model Checking," IEEE 16th International Working Conference on Source Code Analysis and Manipulation 2016.

Wensheng Guo, Yong Wang, Xia Yang and **Min Gao**, "Codecomb: Automated Test Case Generation and Defect Detecting for Embedded Software Based on Symbolic Execution," Journal of Chinese Computer Systems, vol 38 issue 6, 2017.

Min Gao, Kun Wang and Lei He, "Probabilistic Model Checking for Green Energy Router System in Energy Internet, " 2017 IEEE Global Communications Conference.

Min Gao, Kun Wang and Lei He, "Probabilistic Model Checking and Scheduling Implementation of an Energy Router System in Energy Internet for Green Cities," IEEE Transactions on Industrial Informatics, vol. 14, no. 4, pp. 1501-1510, April 2018.

CHAPTER 1

Introduction

1.1 Motivation

With the increasing power of computers and advances in constraint solving technologies, formal and semi-formal verification have received great attentions on many applications. Formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics.

Formal and semi-formal verification have the potential to be applied on lots of applications in real world. Among all the possible applications, one of the most widely used applications of is on software testing. Software testing is a process of analyzing or executing programs with the intent of finding the software bugs. It can also be stated as the process of validating and verifying that the program meets the specifications and requirements during design and maintenance. As one of the most prevalent techniques on software testing, *bounded model checking* (BMC) [CBR01, KCY, CMN12, MFS12] is a classical formal verification technique. Given a program under test and a bound k, BMC unrolls loops and inlines function calls k times to construct an *acyclic* program which is an under-approximation of the original program. It then performs *verification condition* (VC) generation over the acyclic program to obtain a formula which encodes the acyclic program and a property to check. The formula is then fed into a SAT solver. If the formula is proved to be valid by the solver, the property holds. Otherwise, the solver provides a model from which we can extract an execution of the program that violates the property. BMC provides a way to encode and reason about multiple execution paths simultaneously using a single formula, but its scalability is often

limited by deterministic dependencies between program paths and data values.

With the increasing power of computers and advances in constraint solving technologies, an automated dynamic testing technique called concolic testing [GKS05, SMA05] has received great attentions due to its low false positives and high code coverage [CZG13, CS13]. Concolic testing runs a program under test with a random input vector. It then generates additional input vectors by analyzing previous execution paths. Specifically, concolic testing selects one of the branches in a previous execution path and generates a new input vector to steer the next execution toward the opposite branch of the selected branch. By carefully selecting branches for the new inputs, concolic testing avoids generating redundant input vectors that execute the same program path, and thus *enumerates* all non-redundant program paths. One challenge for concolic testing is that it suffers from *path explosion*. It may enumerate exponentially many unique program paths one by one [God07, CZG13, CS13, ABC13], which often leads to poor branch coverage in limited time.

To allieviate this issue and improve concolic testing, we utilize the advantage of BMC, and propose a concolic+BMC algorithm. Given a program under test, the algorithm starts with the per-path search mode in concolic testing while referring to the control flow graph (CFG) of the program to identify easy-to-analyze portions of code that do not contain loops, recursive function calls, or other instructions that are difficult to generate formulas using BMC. Whenever a concolic execution encounters such a portion, the algorithm switches to the BMC mode and generates a BMC formula for the portion, and identifies a frontier of hard-to-analyze instructions. The BMC formula summarizes the effects of all execution paths through the easy-to-analyze portion up to the hard frontier. When the concolic execution reaches the frontier, the algorithm switches back to the per-path search mode to handle the cases that are difficult to summarize by BMC. With 3600 second testing time for each program, our implementation LLSPLAT provides on average 13% relative branch coverage improvement on the programs in the Windows NT drivers simplified set, and on average 16% relative branch coverage improvement on 80 out of 88 programs in the GNU Coreutils set.

In the field of embedded system, reliability of the embedded software has gained more and more interests with the development of the society. Especially in the field of consumer

electronics, industry control, military system and aerospace, huge amount of embedded system modules are playing crucial roles to achieve their functions. Some testing of embedded software are implemented on emulators. There are a few limitations comparing with testing on an emulator and the real embedded platform. First, run-time performance of an emulator is usually slower than real embedded platform. This performance difference is due to the fact that executing instruction in another instruction set such as ARM on top of a x86 machine takes extra work to handle compatibility issue. In practice, even if some emulators are in the form of x86 image such as Android emulator, many applications are not able to run on the emulators. Second, emulators are customized for certain embedded platforms, which is not universal. Even in a specific emulator such as Android emulator, due to the customization on the Android platform from different manufacturers, the emulator may not achieve the same functions as the real customized embedded platform.

As symbolic/concolic testing show good performance on normal software testing, a natural question comes naturally that whether it is possible to deploy dynamic testing methods into real embedded platforms or hardware in the loop testing. We preliminary investigate the possibility of fitting dynamic methods and automatic test case generation into embedded platforms. Since embedded platforms has limited computation and storage resources, we use a client/server model to separate constraint solving and symbolic execution to greatly reduce the computation load in the embedded platform. Experiments on PC and Pandaboard embedded platform preliminary show that Codecomb can run on the embedded platform, and is able to find software deficiencies automatically.

Formal and semi-formal verification is widely applied not only in software field, but also in hardware VLSI design field. VLSI high-level design has many advantages over the commonplace design flow that begins with register-transfer level (RTL) code. One of the most compelling advantages is the improved verification efficiency which a higher level of abstraction offers. It is apparent to the point of being self-evident that when the source code of a design is created, there will be fewer errors if the source is at a higher abstraction level than if it is at a lower level. HLS converts a high-level description of a design into a RTL netlist by considering real design constraint such as area constraints and delay constraints.

It performs instruction scheduling, resource allocation and operation binding on high-level design language such as C into RTL descriptions. As HLS transforms a sequential program into a circuit description that run multiple computations in parallel. The structure and its internal state transition relationship are not guaranteed to remain the same. Thus there is a process needed to verify the the transformations which are applied to the design description. Among all the formal verification categories, functional verification is a useful functionality that can be applied to HLS in a similar manner to its application in RTL-to-gate equivalence checking. Most of functional verification work with formal verification above rely on static analysis on both behaviour and generated RTL side, where model checking is widely used. One issue for the above static analysis methods is that static analysis tries to prove the general equivalence of two state transition machines with full automation, which is hard in general. Another issue is that since model checking describes all the state transitions of a program model in their formulas including non-critical data paths and components that may not affect the output, the number of states grows exponentially along with the increasing scale.

Comparing to static analysis, dynamic testing method such as symbolic execution and concolic execution explores model under investigation on a path-by-path basis. A single path has much lighter encoding than a whole model. It can summarize both combinational and sequential circuit trace accurately. Path based exploration then has the advantage of easily checking all variable state transitions through all paths. It also avoids the loop unrolling problem since it systematically enumerates all paths, and paths can be selected with priority based on verification need. R. Mukherjee *et al.* [MTK16] developed a tool V2C that translates Verilog to C such that dynamic testing for sequential programs can be applied on Verilog. V2C accepts synthesizable Verilog as input and generates a word-level C program as an output. Equivalence checking is then achievable on C level with the help of either static analyzing tools or dynamic execution tools. One major disadvantage of V2C in HLS equivalence checking is that this process needs to verify or to prove the transformation using V2C is equivalent to original Verilog, which is actually a reversed process of HLS functional checking. A formal equivalence proof is needed to show the correctness of the flow. Also

current V2C implementation have difficulties in dealing with dependency of inter-modular combinatorial paths or combinatorial loops. It is hard to determine stability condition for large circuits to obtain an equivalent C program automatically. On the contrary, symbolic execution on top of Verilog directly does not have similar issues as it generates symbolic expressions of the register-transfer relationship precisely. Thus directly applying V2C based framework is not a natural solution for HLS functional verification. As for standalone dynamic execution for Verilog, X. Qin *et al.* [QM14] proposed a concolic execution framework for Verilog similar to [SMA05]. It first instruments the Verilog design, then interleaves concrete and symbolic simulation to obtain execution traces, and then rearranges trace to generate path constraints and concrete testing case. Different from [SMA05], however, this method summarizes symbolic path constraints using the result from concrete simulation, while concolic testing for software has symbolic expression and concrete value generated at the same time. The performance bottleneck of this method lies on the concrete simulator. In fact, as long as here are no constraints unsolvable by the SMT sovler, pure symbolic execution without any concrete simulation can achieve the same functionally. In addition, from HLS equivalence checking perspective, there are some room for optimization in reducing the complexity of SMT solver. Thus far, a full flow of pure dynamic methods for HLS functional verification is still missing.

To fulfill this task, we propose a new HLS functional verification flow via symbolic execution using LegUp [CCA13] as the HLS engine. More precisely, our verification flow first runs light-weight symbolic execution on both C code and HLS generated Verilog code to generate SMT execution trace, respectively. The on-the-fly symbolic execution on Verilog side is implemented on Verilog's SMT representation obtained by Yosys [Wol] synthesis tool. Then we perform optimization on module input symbolization and valid datapath identification from the HLS code generation perspective to greatly reduce the complexity of the SMT solver. Specifically, signals in Verilog traces are collapsed from clock-based encoding to clock-invariant version-based encoding. Considering HLS binds the same operation to an identical set of hardware, we perform one-on-one trace matching by abstracting identical operations on C and Verilog traces such that the scalability issue can be alleviated. Various experiment

results show that our framework has the ability to verify the functional equivalence for numerical computing circuits, and prove the potential of using formal methods for practical HLS functional verification flow for future.

Formal and semi-formal verification have applications other than conventional software and hardware fields as well. One of the application fields is on Energy Internet (EI). Energy crisis and carbon emission have become two seriously concerned issues in green cities [Che07, BZC15, ENS17] recently. As a feasible solution, EI [WYY17, ZYX16] has aroused global concern since it has been proposed. EI is a new power generation developing a green vision of evolution of smart grids into the Internet. Its organization is shown in Fig. 5.1. The key device to compromise EI is energy router (ER) [JWW16]. ER communicates with users similar to an Internet router, thus can perform immediate communication and control according to real-time user status to achieve green efficient energy management.This is vital to realizing green cities [OF15, MTG00]. The design of EI, especially the design of ER based system, requires verification to check all properties related to reliability and economy.

In real world, systems such as ER based systems are inherently probabilistic. Thus quantitative properties are of the greatest interest to verify in addition to logic properties for formal verification. Probabilistic model checking [KNP07] formulates systems into probabilistic transition models such as discrete-time Markov chains (DTMC), continuous-time Markov chains (CTMC), and Markov decision processes (MDP). A quantitative logic property is then applied to the model to check the result, and return a counter-example if a property is not satisfied. We propose a probabilistic model checking method to ER based system design, and monitor ER based system's running behaviour via our scheduling scheme. Specifically, we first propose a CTMC model on an ER based system containing multiple ERs to check the reliability of the system operation. Then we propose a CTMC model on a green ER based subsystem to perform model checking on its reliability and communication count properties. To apply all the communication functions into the real scenario for green cities, we propose an MDP electricity trading model, and model check quantitative properties on the service requester's cost and the service provider's loss. We also propose an energy scheduling simulation scheme for ER based system. In this scheme, we divide a load demand curve into

multiple time windows, and then project each demand in a time window into a cloudlet in cloud computing, and then the energy scheduling process in ER based system is projected to host allocation process in cloud computing. We define our own host allocation policy to complete the scheme.

1.2 Original Contributions

The contributions of this dissertation are summarized as follows. On the applications of formal verification in the software testing field, We propose a concolic+BMC algorithm that applies BMC locally targeting at loop-free code fragment during concolic testing to alleviate path explosion, and thus improve branch coverage. To the best of out knowledge, it is the first time that static analysis methods such as BMC has been used to speed up concolic execution. We also show the an real implementation to perform dynamic analysis method into embedded platforms.

On the applications of formal verification in the VLSI HLS field, we propose a pure symbolic execution flow to tackle functional verification of HLS. To the best of our knowledge, it is the first time that a pure dynamic approach is applied on HLS functional verification without transforming one side of the source code to the format of its counterpart before verification. We also present a Verilog pure symbolic execution framework without performing concrete simulation that runs directly from Verilog's SMT representation. In addition, we perform optimization on module input symbolization from HLS code generation perspective and clock-invariant version-based encoding to greatly reduce the complexity for SMT solver. Finally, we introduce an operation abstraction method to alleviate the scalability issue considering allocated hardware usage from HLS along the datapath.

On the applications of formal verification in the Energy Internet field, we introduce CTMC and MDP state machines to model ER based systems. To the best of out knowledge, it is the first time that formal verification technique is applied to ER based systems. Moreover, we project the energy scheduling of ER based system into cloud computing area, and implement a tool to observe the performance of ER based systems due to the similarity of these two

areas. It is the first time that a cloud computing tool is tailored to suit the need for ER based systems. Finally, we consider both electricity price and line loss during power transmission during the selection of power service providers. Extensive experiment verifies the effectiveness of the proposed models and the monitoring scheme.

1.3 Organization

The organization of this dissertation is shown as follows. In Chapter 2, we present LLSPLAT, a framework to improve concolic testing by bound model checking. In Chapter 3, we study the possibilities of employing dynamic testing methods to embedded platforms, and introduce the implementation Codecomb. In Chapter 4, we show how dynamic testing methods are applied in VLSI field to check the functional equivalence of HLS. The formal modeling and simulation implementation of the energy router based system in RI is described in Chapter 5. Finally, we conclude the dissertation in Chapter 6.

CHAPTER 2

LLSPLAT: Improving Concolic Testing by Bounded Model Checking

2.1 Introduction

With the increasing power of computers and advances in constraint solving technologies, an automated dynamic testing technique called concolic testing [GKS05, SMA05] has received much attention due to its low false positives and high code coverage [CZG13, CS13]. Concolic testing runs a program under test with a random input vector. It then generates additional input vectors by analyzing previous execution paths. Specifically, concolic testing selects one of the branches in a previous execution path and generates a new input vector to steer the next execution toward the opposite branch of the selected branch. By carefully selecting branches for the new inputs, concolic testing avoids generating redundant input vectors that execute the same program path, and thus *enumerates* all non-redundant program paths. In practice, concolic testing suffers from *path explosion*: it may enumerate exponentially many unique program paths one by one [God07, CZG13, CS13, ABC13], which often leads to poor branch coverage in limited time.

On the other hand, *bounded model checking* (BMC) [CBR01, KCY, CMN12, MFS12] is a fully symbolic testing technique. Given a program under test and a bound k, BMC unrolls loops and inlines function calls k times to construct an *acyclic* program which is an under-approximation of the original program. It then performs *verification condition* (VC) generation over the acyclic program to obtain a formula which encodes the acyclic program and a property to check. The formula is then fed into a SAT solver. If the formula is proved to be valid by the solver, the property holds. Otherwise, the solver provides a model from

which we can extract an execution of the program that violates the property. BMC provides a way to encode and reason about multiple execution paths simultaneously using a single formula, but its scalability is often limited by deterministic dependencies between program paths and data values.

A natural question is *whether there is a way to improve concolic testing by BMC to alleviate path explosion and thus improve branch coverage*? In this paper, we provide a positive answer and propose a concolic+BMC algorithm. Intuitively, given a program under test, the algorithm starts with the per-path search mode in concolic testing while referring to the control flow graph (CFG) of the program to identify easy-to-analyze portions of code that do not contain loops, recursive function calls, or other instructions that are difficult to generate formulas using BMC. Whenever a concolic execution encounters such a portion, the algorithm switches to the BMC mode and generates a BMC formula for the portion, and identifies a frontier of hard-to-analyze instructions. The BMC formula summarizes the effects of all execution paths through the easy-to-analyze portion up to the hard frontier. When the concolic execution reaches the frontier, the algorithm switches back to the per-path search mode to handle the cases that are difficult to summarize by BMC.

We have implemented the concolic+BMC algorithm on top of KLEE [CDE08] and called the new tool Llsplat. We have compared Llsplat with KLEE, using 10 programs from the Windows NT Drivers Simplified [SVC] and 88 programs from the GNU Coreutils used in [CDE08]. With 3600 second testing time for each program, Llsplat provides on average 13% relative branch coverage improvement on the programs in the Windows NT drivers simplified set, and on average 16% relative branch coverage improvement on 80 out of 88 programs in the GNU Coreutils set.

The rest of the paper is organized as follows. Section 2 provides a motivating example. Section 3 reviews concolic testing. Section 4 describes the concolic+BMC algorithm. Section 5 presents experimental results. Section 6 shows related work. Section 7 concludes this paper.

2.2 A Motivating Example

We illustrate the inadequacy of concolic testing, and the benefits of using BMC to improve concolic testing, using the function `foo` below. The function runs in an infinite loop, and receives two inputs in each iteration. One input `c` is a character and the other input `s` is a character array. The function `foo` reaches the label L if the variable `state` is 9, and the input array `s` holds the string "reset". From the label L, there is a huge chunk of code consisting conditionals, loops, and procedure calls. Similar functions like `foo` are often generated by lexers.

Concolic testing systematically explores all execution paths of the function. Since the function `foo` runs in an infinite loop, the number of distinct feasible executions is infinite. To perform concolic testing we need to bound the number of iterations of the loop if we perform a depth-first search of the execution paths. There are 17 possible choices of values of `c` and `s` that concolic testing would consider, and at least 9 iterations are required to reach the label L. Hence, concolic testing will explore about $17^9 \approx 10^{11}$ execution paths. It is unlikely that concolic testing can explore a path that reaches the label L and executes the code below the label L in a reasonable time budget. We confirm this fact by testing the function `foo` using KLEE [CDE08]. It could not reach the label L in a day, which led to poor branch coverage.

It is worth mentioning that, if there were buggy code after the label L, the situation would get even worse because concolic testing cannot reveal the bugs efficiently.

```
1  void foo() {
2    char c, s[6];
3    int state = 0;
4
5    while(1) {
6      ... // Some dummy code
7      c = input(); s = input();
8
9      if (c == '[' && state == 0) state = 1;
```

```
10      if (c == '(' && state == 1) state = 2;
11      if (c == '{' && state == 2) state = 3;
12      if (c == '~' && state == 3) state = 4;
13      if (c == 'a' && state == 4) state = 5;
14      if (c == 'x' && state == 5) state = 6;
15      if (c == '}' && state == 6) state = 7;
16      if (c == ')' && state == 7) state = 8;
17      if (c == ']' && state == 8) state = 9;
18      if (s[0] == 'r' && s[1] == 'e' && s[2] == 's' &&
19          s[3] == 'e' && s[4] == 't' && s[5] == 0 &&
20          state == 9)
21        goto L;
22    }
23  L: ... // A large chunk of code below.
24  }
```

In our concolic+BMC approach, whenever a concolic execution encounters a conditional, it has a choice either to save a predicate representing that a particular branch is taken along the execution as concolic testing does, or to save a BMC formula, for example, that encodes the entire conditional. Which choice is taken depends on whether the conditional is "simple" enough to generate a BMC formula easily. For example, a conditional is simple if there are no loops and recursive function calls[1] inside it. Since all conditionals are simple in function `foo`, the concolic+BMC approach can easily generate a feasible path that reaches the label L and thus greatly shorten the time to reach the subsequent uncovered paths. We validated this fact by using LLSPLAT to test function `foo`. LLSPLAT could generate paths that reached the label L in 3s, which led to better branch coverage.

[1]The program size after function inlining can be exponentially larger than the size of the original program.

$$
\begin{array}{lll}
P & ::= & (\text{var } g)^* \cdot Fn^+ \\
Fn & ::= & f((\text{var } p)^*) \cdot (\text{var } l)^* \cdot BB^+ \\
BB & ::= & Inst^* \cdot TermInst \\
Inst & ::= & x \longleftarrow e \mid f(e^*) \mid x \longleftarrow input() \\
TermInst & ::= & ret \mid br\ e\ BB1\ BB2 \mid br\ BB \mid ERROR
\end{array}
$$

Figure 2.1: Program Model

2.3 Concolic Testing

We first review the traditional concolic testing algorithm with depth-first path searching stratergy, and then describe how LLSPLAT modifies it.

2.3.1 Program Model

We describe how concolic testing works on a simple language shown in Figure 2.1. A *program* consists of a set of *global* variables and a set of *functions*. Each function consists of a name, a sequence of formal parameters, a set of *local* variables, and a set of *basic blocks* representing the *control flow graph* (CFG) of the function. Each basic block consists of a list of *instructions* followed by a *terminating instruction*. There are three types of instructions: $x \longleftarrow e$ is an assignment, $f(e^*)$ is a function call, and $x \longleftarrow input()$ indicates that the variable x is a program input. There are four types of terminating instructions: ret is a return instruction, $br\ e\ BB1\ BB2$ is a conditional branch, $br\ BB$ is an unconditional branch, and $ERROR$ indicates program abortion. We omit an explicit syntax of expressions. We assume there is an entry function `main` that is not called anywhere. We also assume each function has an *entry* basic block, and every basic block of the function is reachable from it.

2.3.2 The Concolic Testing Algorithm

To test a program P, concolic testing tries to explore all execution paths of P. It first instruments the program P by Algo 1, and outputs an instrumented program P'. The red-highlighted lines(gray with monochrome printers) in the algorithms can be ignored for now

because they are used in the concolic+BMC approach we describe later. Algo 2 repeatedly runs the instrumented program P'. Due to limited space, we omit the instrumentation for function calls, and the code that bounds the search depth in the search strategy — these are identical to previous work [GKS05, SMA05].

Algo 1 first makes a copy P' of the program P, and inserts various global variables and function calls which are used for the symbolic execution. It then returns the instrumented program P'. Algo 3 presents the definitions of the instrumented functions. The expressions enclosed in double quotes ("e") represent syntactic objects. We denote $\&x$ to be the address of a variable x.

Algorithm 1: Instrumentation

```
Program instrument(P):
    P' ⟵ P
    Add to P' global vars i ⟵ 0, inputNo ⟵ 0, symStore ⟵ [], pathC ⟵ []
    Govs ⟵ {BB | BB is a governor in P}
    Add to P' global vars bmcNo ⟵ 0, currGov ⟵ None, init ⟵ None
    foreach BB ∈ P' do
        if BB ∈ GR(gov) for some gov ∈ Govs then continue
        foreach Inst ∈ BB do
            switch Inst do
                case x ⟵ input() do
                    Replace Inst by InitInput("x")
                case x ⟵ e do
                    Add updateSymStore("x", "e") before Inst
                case br e BB1 BB2 do
                    if BB ∈ Govs then
                        Add startBMC(BB) before Inst
                        foreach d ∈ Dests(BB) do
                            Add endBMC(BB, d) as the 1st instruction of d
                    else
                        Add addPathConstraint("e", e) before Inst
                case Return do
                    if Inst is in the main function then
                        Add solveConstraint() before Inst
                case ERROR do
                    Add print("ERROR found") before Inst
    return P'
```

Algorithm 2: run_llsplat

```
void run_llsplat(P):
    I ⟵ []; branch_hist ⟵ []; completed ⟵ false
    CFG_P ⟵ CFGofProgram(P)
    while ¬completed do execute instrument(P)
```

The function $InitInput(\text{“}x\text{”})$ initializes the input variable x using the *input map* I in all runs except the first. The variable x is assigned randomly in the first run. The function also saves a fresh symbolic variable for x in the symbolic store.

The function $updateSymStore(\text{“}x\text{”}, \text{“}e\text{”})$ updates x's symbolic expression in the symbolic store $symStore$ based on the expression e. We write $symexpr(\text{“}e\text{”})$ to represent the symbolic expression by substituting each variable v in “e” with its symbolic expression $symStore[\&v]$. For example, if “e” = “$a + b$”, $symStore[\&a] = e_a$, and $symStore[\&b] = e_b$, then $symexpr(\text{“}e\text{”}) = e_a + e_b$.

The function $addPathConstraint(\text{“}e\text{”}, e)$ updates the *path constraint* $pathC$ and the *coverage history* $branch\_hist$. Symbolic predicate expressions from the branching points are collected in the list $pathC$. At the end of the execution, $pathC$ contains all predicates whose *conjunction* holds for the execution path. To explore paths of the program under test, each run (except the first) is executed based on the coverage history computed in the previous run. The coverage history is a list of $BranchNode$s. A $BranchNode$ has two boolean fields: $isCovered$ records which branch is taken, and *done* records whether both branches have executed in prior runs (with the same history up to this branch node).

The function $solveConstraint()$ determines new inputs that forces the next run to execute the last unexplored branch of the j-th conditional in $branch\_hist$.

2.4 Combining Concolic Testing with BMC

Recall that the goal of our work is that, whenever a concolic run starts with some inputs, instead of using the path constraint to encode a single execution path, we plan to add BMC formulas to the path constraint so that it may encode (potentially exponentially) many execution paths. Note that the path constraint is used by a constraint solver to decide new

Algorithm 3: Concolic Testing

```
void InitInput("x"):
    inputNo ⟵ inputNo + 1
    j ⟵ inputNo
    if I[j] is undefined then
        x ⟵ random()
        I[j] ⟵ x
    else
        x ⟵ I[j]
    symStore[&x] ⟵ sym_j
void updateSymStore("x", "e"):
    symStore[&x] ⟵ symexpr("e")
struct BrNode:
    isCovered : bool
    done : bool
void SolveConstraint():
    j = i - 1
    while j ≥ 0 do
        if ¬br_hist[j].done then
            if br_hist[j] is BmcNode then
                foreach d such that ¬br_hist[j].isCovered[d] do
                    if ⋀_{0≤k≤j−1} pathC[k] ∧ rmLastDest(path_c[j]) ∧ ⋁_{c∈Edges_d[d]} c has a solution I′ then
                        br_hist ⟵ br_hist[0..j]
                        I ⟵ I′
                        return
            else
                br_hist[j].isCovered ⟵ ¬br_hist[j].isCovered
                pathC[j] ⟵ ¬pathC[j]
                if pathC[0..j] has a solution I′ then
                    br_hist ⟵ br_hist[0..j]
                    I ⟵ I′
                    return
        j = j − 1
    if j < 0 then completed ⟵ true
void addPathConstraint("e", b):
    if b then
        pathC[i] ⟵ symexpr("e")
    else
        pathC[i] ⟵ ¬symexpr("e")
    if i < |br_hist| then
        if i = |br_hist| − 1 then
            br_hist[i].done ⟵ true
    else
        br_hist[i] ⟵ BrNode(isCovered : b, done : false)
    i ⟵ i + 1
```

runs in concolic testing. When we attempt to cover an uncovered instruction in the new run, the path constraint that encodes multiple execution paths enables the constraint solver to search for multiple paths leading to the instruction, instead of one path in the traditional concolic testing.

To achieve the goal, given a program under test, we identify *regions* of the program for generating BMC formulas. A region of a program is a subgraph of the control flow graph of

the program. We list the following requirements for identifying regions.

1. A region must be acyclic. It is required by any BMC procedure.

2. A region should not have function calls. It is desired because function inlining is required before generating BMC formula but the resulting BMC formula may be exponentially large in the size of the input program, which we want to avoid.

3. A region should be sufficiently large so that the corresponding BMC formula covers more paths and fully exploits the power of a modern constraint solver. Most of the path constraints in those multiple paths are the same except for the constraints within the BMC regions, thus solving one concolic+BMC constraints representing multiple paths may take less time than solving multiple independent paths.

In addition, given a region, it is also required that a desired BMC generation procedure be *compatible* with concolic testing. Unlike generating a formula for an entire program in existing BMC tools, we need to generate ones for regions of a program which lead to some specific issues, which BMC procedures in existing BMC tools need not and cannot handle. For example, a natural question would be that, after adding a BMC formula to the path constraint, how the symbolic store needs to be updated so that concolic testing proceeds.

We now present the concolic+BMC algorithm. The key observation is that given a program P under test, the instrumented program for P can additionally refer to the (static) CFG of P and perform static analysis at run time. Section 2.4.1 describes how to identify program portions for BMC formula generation. Section 2.4.2 describes the BMC formula generation algorithm. Section 2.4.3 integrates this with concolic testing.

2.4.1 Identifying Program Portions for BMC

2.4.1.1 Preliminaries

Given a CFG, a basic block m *dominates* a basic block n if every path from the entry basic block of the CFG to n goes through m. We denote $Dom(m)$ to be the set of basic blocks

which m dominates. A depth-first search of the CFG forms a *depth-first spanning tree* (DFST). There are edges in CFG that go from a basic block m to an ancestor of n in DFST (possibly to m itself). We call these edges *back edges*. By [CSR01], a directed graph is acyclic if a depth-first search yields no back edge.

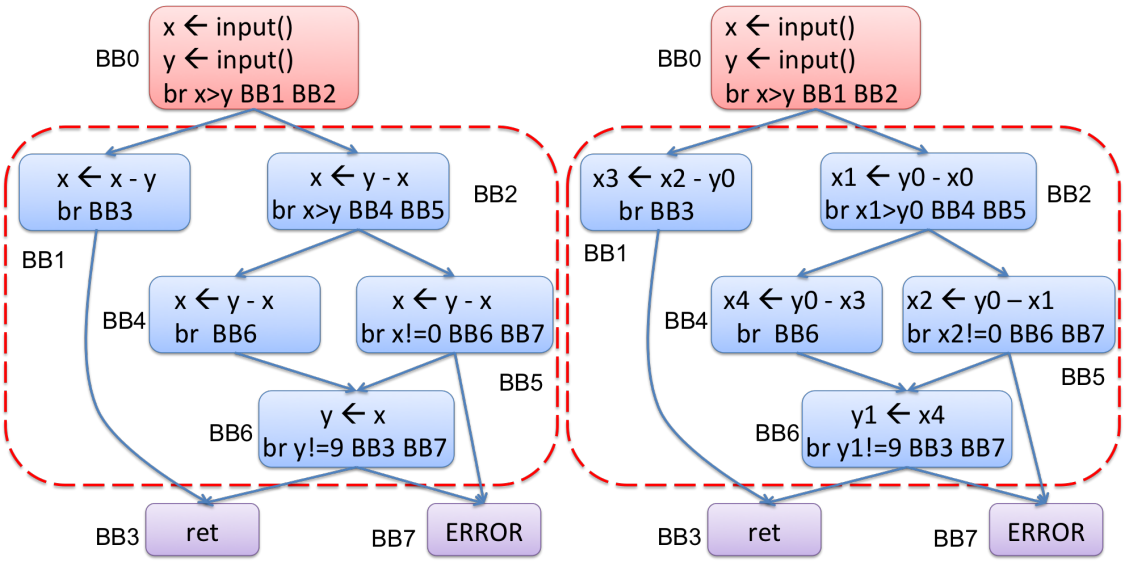
2.4.1.2 Governors, Governed Regions, and Destinations

Given a basic block m, a basic block $n \in Dom(m)$ is *polluted* in $Dom(m)$ in the following four cases: (1) n contains function call instructions, (2) n has no successors, (3) n is the source or the target of a back edge, or (4) n is reachable from a polluted basic block $k \in Dom(m)$. A basic block m *effectively dominates* a basic block n if $n \in Dom(m)$ and n is not polluted in $Dom(m)$. We denote $Edom(m)$ to be the set of basic blocks that m effectively dominates.

A basic block m is called a *governor candidate* if (1) the terminating instruction of m is of the form *br e* $BB1$ $BB2$, (2) m dominates both $BB1$ and $BB2$, and (3) $Edom(BB1)$ and $Edom(BB2)$ are not empty. Given a governor candidate m with its two successors $BB1$ and $BB2$, the *governed region* of m, denoted by $GR(m)$, is $Edom(BB1) \cup Edom(BB2)$. A basic block n is a *destination* of $GR(m)$ if $n \notin GR(m)$ and n is a successor of some basic block $k \in GR(m)$. Let the set $Dests(m)$ be all destinations of $GR(m)$. A basic block *gov* is a *governor* if *gov* is a governor candidate, and there is no governor candidate m with $gov \in GR(m)$. For any governor *gov*, $GR(gov)$ is acyclic and does not have function calls, and *gov* dominates every basic block $BB \in GR(gov)$.

2.4.1.3 Example

Consider the program in Fig 2.2a. $BB0$ is a governor. Its governed region $GR(BB0)$ includes $BB1$, $BB2$, $BB4$, $BB5$, and $BB6$, which are inside the red dash circle. $BB3$ and $BB7$ are the destinations in $Dests(BB0)$. Though $BB2$ is a governor candidate, it is not a governor because it is in $GR(BB0)$.



(a) A program under test (b) Variable renaming

Figure 2.2: An Example

2.4.2 Translating Governed Regions to BMC Formulas

A governed region is ideal for generating a BMC formula because it is acyclic, does not have function calls, and is "sufficiently" large in the sense that it includes as many (unpolluted) basic blocks as its governor governs. We present our algorithm that translates a governed region to a BMC formula, and provide an example.

2.4.2.1 The BMC Formula Generation Algorithm

Given a governor gov, we construct a BMC formula ϕ for $GR(gov)$ in five steps:

1. Renaming variables in $GR(gov)$ into an SSA-form. Since $GR(gov)$ is acyclic, there exists a topological ordering over the basic blocks in $GR(gov)$. Let $AccVars$ be the set of variables accessed by the instructions in $GR(gov)$, and let a *version map* $\mathcal{V}$ be a map from each variable $x \in AccVars$ to a variable x_α with a version $\alpha \in \mathbb{N}$. We assign the version number α to each variable in $GR(gov)$ for renaming according to the topological order.

2. Create a boolean variable g_{BB} for each basic block $BB \in GR(gov)$, which is called an

execution guard. Our intention is that, if g_{BB} is true, then an execution represented by a model of the final BMC formula ϕ goes through BB. Otherwise, it does not goes through BB.

3. Compute an *edge map* $Edges$ that maps each basic block $BB \in GR(gov) \cup Dests(gov)$ to a list of *edge formulas*. Each entry in the list represents the condition of hitting an incoming edge of a basic block. For each $BB \in GR(gov)$, if its terminating instruction is $br\ e\ BB1\ BB2$, then we add $g_{BB} \wedge e$ to $Edges[BB1]$, and add $g_{BB} \wedge \neg e$ to $Edges[BB2]$; if it is $br\ BB1$, then we add g_{BB} to $Edges[BB1]$. If the terminating instruction belongs to a governor, we insert the condition pair e and $\neg e$ to the corresponding $Edges$ map directly. Let the governor's terminating instruction be $br\ e\ BB1\ BB2$. Let e_0 be an expression obtained by replacing each variable x in e with x_0. We set $Edges[BB1] = e_0$ and $Edges[BB2] = \neg e_0$.

4. Compute a *block map* $Blks$ that maps each basic block $BB \in GR(gov)$ to a *block formula*. For each $BB \in GR(gov)$, let $I_1, I_2, \ldots, I_k$ be the non-terminating instructions in BB. For each $1 \leq i \leq k$, if I_i is $x_\alpha \longleftarrow e$, we define an *instruction formula* c_i to be $x_\alpha = ite(g_{BB}, e, x_{\alpha-1})$. We set $Blks[BB] = \bigwedge_{1 \leq i \leq k} c_i$.

5. Create the final BMC formula ϕ, defined as follows:
$$\bigwedge_{BB \in GR(gov)} \left(\left(g_{BB} = \bigvee_{c \in Edges[BB]} c \right) \wedge Blks[BB] \right)$$
Intuitively, ϕ claims that for each basic block $BB \in GR(gov)$, (1) BB is taken (i.e., g_{BB} is true) if one of its predecessor is taken, and (2) the block formula of BB must hold.

Our BMC formula generation algorithm has the following important property.

Theorem 2.4.1. *Let gov be a governor and T be an arbitrary topological ordering over $GR(gov)$. After the BMC algorithm is done w.r.t. T, for any destination $d \in Dests(gov)$, (1) the formula $\phi \wedge \bigvee_{c \in Edges[d]} c$ encodes all executions from gov to d, and (2) for every execution ρ from gov to d, the final version of each variable x in ϕ represents the value of x when ρ enters d.*

BB	$Edges[BB]$	$Blks[BB]$
$BB1$	$\{x0 > y0\}$	$x3 = ite(g_{BB1}, x2 - y0, x2)$
$BB2$	$\{\neg(x0 > y0)\}$	$x1 = ite(g_{BB2}, y0 - x0, x0)$
$BB3$	$\{g_{BB1}, g_{BB6} \wedge y1 \neq 9\}$	
$BB4$	$\{g_{BB2} \wedge x1 > y0\}$	$x4 = ite(g_{BB4}, y0 - x3, x3)$
$BB5$	$\{g_{BB2} \wedge \neg(x1 > y0)\}$	$x2 = ite(g_{BB5}, y0 - x1, x1)$
$BB6$	$\{g_{BB4}, g_{BB5} \wedge x2 \neq 0\}$	$y1 = ite(g_{BB6}, x4, y0)$
$BB7$	$\{g_{BB5} \wedge \neg(x2 \neq 0), g_{BB6} \wedge \neg(y1 \neq 9)\}$	

Table 2.1: Edge formulas and block formulas

Example We illustrate our BMC algorithm by reusing the example in Fig 2.2a. The topological order we use for the variable renaming is $BB2, BB5, BB1, BB4, BB6$. After variable renaming, the resulting program is in Fig 2.2b. After Step 4 of the algorithm, the edge map $Edges$ and the block map $Blks$ are shown in Table 2.1.

To give a flavor of the correctness of Theorem 2.4.1(2), we examine an execution ρ : $BB0, BB1, BB3$ as an example. When ρ enters the destination $BB3$, the largest version of x and y along ρ is $x3$ and $y0$, but their final versions in ϕ are $x4$ and $y1$. However, since $BB2, BB4, BB5$ and $BB6$ are not taken along ρ, we have $x4 = x3$, $x2 = x1 = x0$, and $y1 = y0$. Since $BB1$ is taken, we have $x3 = x2 - y0$. Thus $x4 = x0 - y0$ and $y1 = y0$. We conclude that $x4$ and $y1$ represent the values of x and y when ρ enters the destination $BB3$.

2.4.3 Integrating BMC Formulas with Concolic Testing

To integrate BMC with concolic testing, we need to instrument function calls for BMC encoding, obtain program CFG for performing static analysis, and consider the way to update concolic data structures such as the worklist, the symbolic store and the branch history. Thus we add the red lines in Algo 1, 2, and 3 to reflect the integration.

Once we meet a branch instruction, we need to check whether this instruction is a good candidate to perform BMC. During the instrumentation in Algo 1, we first compute a set $Govs$ of all governors of the program P. Since basic blocks in governed regions are used

to generate BMC formulas, we skip instrumenting them. When a basic block BB has two successors, if BB is a governor, we instrument a function call $startBMC(BB)$ before BB's terminating instruction, and for each destination $d \in Dests(BB)$, we instrument a function call $endBMC(BB, d)$ as the first instruction of d. If BB is not a governor, we perform the old instrumentation in concolic testing.

BMC formulas generation in our algorithm requires static analysis result of a program under test. In Algo 2, we read the CFG of the uninstrumented program P. This CFG is used to generate BMC formulas along concolic executions.

Algorithm 4: startBMC and endBMC

```
void startBMC(gov):
    currGov ⟵ gov; bmcNo ⟵ bmcNo + 1
    init ⟵ ⋀_{x∈AccVars(gov)} (x_0^{bmcNo} = symStore[&x]) // x_0^{bmcNo} is a fresh variable
struct BmcNode:
    isCovered : BasicBlock ⟶ bool
    Edges_d : BasicBlock ⟶ formula
    done : bool
void endBMC(gov, d):
    if currGov ≠ gov then return
    (φ, V_final, Edges) ⟵ doBMC(CFG_P, gov)
    if i <| branch_hist | then
        if i =| branch_hist | −1∧∀d' ∈ Dests(gov)\{d}. branch_hist[i].isCovered[d'] then
            branch_hist[i].done ⟵ true
    else
        branch_hist[i] ⟵ BmcNode(isCovered : λdest ∈ Dests(gov). false,
            Edges_d : λdest ∈ Dests(gov). addSup(Edges[dest], bmcNo), done : false)
    branch_hist[i].isCovered[d] ⟵ true
    pathC[i] ⟵ init ∧ addSup(φ ∧ ⋁_{c∈Edges[d]} c, bmcNo)
    i ⟵ i + 1
    foreach x ∈ AccVars(gov) do SymStore[&x] ⟵ addSup(V_final(x), bmcNo)
```

Since a BMC region may contain multiple destinations, we need to instrument one $startBMC(gov)$ call and multiple $endBMC(gov, d)$ calls into the program to define a BMC governed region.

The definition of $startBMC(gov)$ is given in Algo 4. It saves the governor gov that will be used to generate a BMC formula using $currGov$. Then it increments $bmcNo$, which records the number of BMC formulas that have been generated so far along the concolic

execution. It then uses *init* to "glue" the execution before entering $GR(gov)$ with the BMC formula for $GR(gov)$. More concretely, for each variable $x \in AccVars(gov)$, an equation $x_0^{BmcNo} = symStore[\&x]$ is created, and *init* is the conjunction of all such equations.

The definition of $endBMC(gov, d)$ is also given in Algo 4. If the passed-in governor *gov* is the one saved in *currGov*, it performs the BMC generation algorithm described in Section 2.4.2 to obtain a BMC formula ϕ for the governed region $GR(gov)$, the final version map $\mathcal{V}_{final}$, and the edge map *Edges*.

The coverage history *branch_hist* is updated in $endBMC(gov, d)$. We extend *branch_hist* to be a list of $BranchNode \cup BmcNode$. A *BmcNode* has three fields: *isCovered* records which destinations have been covered in prior runs, *Edges_d* maps each destination to its edge formulas, and *done* records whether all destinations have been covered in prior runs.

We then start to integrate BMC formulas into concolic testing. Given a formula ψ and a number j, we denote $addSup(\psi, j)$ to be the formula obtained by replacing each variable x in ψ with a new variable x^j. We first create a formula $\phi \wedge \bigvee_{c \in Edges[d]} c$ which represents all executions from the governor *gov* to the destination d by Theorem 2.4.1. Since the governed region may be reached multiple times along an execution, we compute a formula $\psi \equiv addSup(\phi \wedge \bigvee_{c \in Edges[d]} c, bmcNo)$ which specifies that ψ is the $bmcNo$-th BMC formula along the execution. We then add $init \wedge \psi$ to the path constraint. Finally, to let the concolic execution proceed, for each variable $x \in AccVars(gov)$, we update the symbolic store so that $symStore[\&x]$ represents the value of x when the execution enters the destination d.

The function *SolveConstraint* is extended as shown in Algo 3. If the node $branch\_hist[j]$ is a *BmcNode*, we find an uncovered destination d, and asks if there is an execution that goes to d. The formula $rmLastDest(pathC[j])$ is defined by removing the disjunction of edge formulas of d' from $pathC[j]$ where d' is the destination covered by the just terminating execution. If there are new inputs I' for such an execution to d, a new run is started with inputs I'.

Example We again reuse the example in Fig 2.2a. Suppose LLSPLAT randomly generates $x = 10$ and $y = 5$ in the first run. When the run terminates, the path constraint is of size 1,

and $pathC[0] = init \wedge \phi \wedge \psi_d$, defined as follows. Note that the superscript 1 of the variables in $pathC[0]$ represents that it is the first BMC formula generated along the run. The symbolic variables $sym1$ and $sym2$ are created for x and y when $InitInput(\text{“}x\text{”})$ and $InitInput(\text{“}y\text{”})$ are called.

$$init \equiv x0^1 = sym1 \wedge y0^1 = sym2$$

$$\phi \equiv \left[\begin{array}{l} g^1_{BB1} = x0^1 > y0^1 \wedge \\ g^1_{BB2} = \neg(x0^1 > y0^1) \wedge \\ g^1_{BB4} = (g^1_{BB2} \wedge x1^1 > y0^1) \wedge \\ g^1_{BB5} = (g^1_{BB2} \wedge \neg(x1^1 > y0^1)) \wedge \\ g^1_{BB6} = (g^1_{BB4} \vee (g^1_{BB5} \wedge g^1_{BB2} \neq 0)) \end{array}\right] \bigwedge$$

$$\left[\begin{array}{l} x3^1 = ite(g^1_{BB1}, x2^1 - y0^1, x2^1)) \wedge \\ x1^1 = ite(g^1_{BB2}, y0^1 - x0^1, x0^1)) \wedge \\ x4^1 = ite(g^1_{BB4}, y0^1 - x3^1, x3^1)) \wedge \\ x2^1 = ite(g^1_{BB5}, y0^1 - x1^1, x1^1)) \wedge \\ y1^1 = ite(g^1_{BB6}, x4^1, y0^1)) \end{array}\right]$$

$$\psi_d \equiv g^1_{BB1} \vee (g^1_{BB6} \wedge y1^1 \neq 9)$$

The coverage history $branch\_hist$ is of size one. $branch\_hist[0]$ is a $BmcNode$ defined below:

$$\begin{array}{l} branch\_hist[0].isCovered = \\ \quad [BB3 \mapsto true, BB7 \mapsto false] \\ branch\_hist[0].done = false \\ branch\_hist[0].Edges\_d = \\ \quad [BB3 \mapsto \{g^1_{BB1}, g^1_{BB6} \wedge y1^1 \neq 9\}, \\ \quad BB7 \mapsto \{g^1_{BB5} \wedge \neg(x2^1 \neq 0), g^1_{BB6} \wedge \neg(y1^1 \neq 9)\}] \end{array}$$

Now LLSPLAT searches for new inputs for the next run. Since $BB7$ is the only uncovered

destination based on $branch\_hist[0].isCovered$, LLSPLAT solves the formula $init \wedge \phi \wedge \bigvee_{c \in branch\_hist[0].Edges\_d[BB7]} c$, that is, LLSPLAT tries to find a feasible execution path that leads to $BB7$ containing $ERROR$. Note that there are three execution paths to $BB7$, and the formula encodes all.

2.5 Experiments

We have built our tool LLSPLAT to implement the concolic+BMC algorithm on top of KLEE (LLVM version 3.4). To verify whether the algorithm can increase branch coverage of concolic testing in practice, we designed our experiments to compare the branch coverage between LLSPLAT and KLEE.

2.5.1 Experiment Settings

We chose two sets of benchmarks to perform the comparison. The first benchmark set is the Windows NT Drivers Simplified set containing 10 C programs from [SVC], and the second benchmark set is the GNU Coreutils tested in [CDE08] that contains 88 C programs. Each program in Windows NT Drivers Simplified set ranges between 2344 and 6444 lines of LLVM-IR code, and that of the GNU Coreutils set contains approximately 200,000 lines of code in LLVM-IR level per benchmark including library code. All the experiments were performed on a 2 core Intel Xeon E5-2667 v2 CPU machine with 256GB memory and 64-bit Linux (Debian/Jessie).

Both LLSPLAT and KLEE ran with original KLEE arguments. We conducted the experiments 10 times and then calculated the average coverage because the coverage depends on the initial random input vector.

All the benchmarks in the Windows NT Drivers Simplified set are tested by calling LLSPLAT and KLEE with maximum run time of 3600s and maximum memory of 1600MB. All the benchmarks in the GNU Coreutils set are tested using the following command

```
./<tool-name> --libc=uclibc
```

```
--posix-runtime
--no-output
--max-memory=1600
--max-time=3600
./file_under_test
--sym-args 1 10 2
--sym-files 2 8
```

Using these options, we ran each benchmark that contained a minimum of 1 argument, and a maximum 10 of arguments with each argument containing at most 2 characters for 3600s.

2.5.2 Experimental Results

Table 2.2 shows the branch coverage of all 10 benchmarks in the Windows NT Drivers Simplified benchmark set. For example, KLEE covered 270 branches while LLSPLAT covered 319 branches with relative branch coverage improvement of 18.14% on *cdaudio_s1_f*. Relative branch coverage improvement is defined by the difference between the branch coverage of LLSPLAT and the branch coverage of KLEE divided by the branch coverage of KLEE.

The results in Table 2.2 show that LLSPLAT achieves better branch coverage than KLEE for all 10 benchmarks in the set with 13% relative branch coverage improvement. We conclude that the concolic+BMC algorithm improves the branch coverage of this benchmark set.

The programs in the GNU Coreutils benchmark set are larger than that of the Windows NT Driver Simplified benchmark set. We would like to evaluate them to check whether LLSPLAT can still achieve higher branch coverage than KLEE.

Figure 2.3 shows the result of relative branch coverage improvement for all 88 programs from the GNU Coreutils benchmark set where each bar represents relative branch coverage improvement of one benchmark. A bar above zero indicates by how much LLSPLAT wins over KLEE; a bar below shows the opposite. Bars are sorted in ascending order.

The result shows that LLSPLAT outperforms KLEE in terms of branch coverage on most of the benchmarks: 80 out of 88 benchmarks tested with LLSPLAT have higher branch coverage

Benchmark	KLEE		LLSPLAT	
	Branch covered	Run time(s)	Branch covered	Run time(s)
cdaudio_s1_f	270	2.26	319 (+18.14%)	61.06
cdaudio_s1_t	268	2.37	317 (+18.28%)	61.46
diskperf_s1_t	114	3614.35	132 (+15.78%)	3602.78
floppy_s3_f	142	1.55	156 (+9.85%)	20.92
floppy_s3_t	142	1.57	155 (+9.15%)	22.36
floppy_s4_f	224	3.60	238 (+6.25%)	31.99
floppy_s4_t	224	3.57	236 (+5.35%)	33.80
kbfiltr_s1_t	93	0.42	111 (+19.35%)	1.62
kbfiltr_s2_f	159	1.11	181 (+13.83%)	4.98
kbfiltr_s2_t	159	1.11	182 (+14.47%)	4.58

Table 2.2: Branch coverage comparison between LLSPLAT and KLEE on the Windows NT Drivers Simplified

than KLEE. Average improvement for all benchmarks is 13% and 16% for the 80 improved benchmarks. Among all the benchmarks 65 have more than 10% increases. Table 2.3 provides detailed branch coverage information of 10 randomly selected benchmarks. The number of encoded BMC governed regions is also provided in Table 2.3 to show the involvement of BMC encoding in this benchmark set. Note that a governed region can be encoded multiple times since it can be reached multiple times during one single execution path.

A natural question that arises is when LLSPLAT's branch coverage exceeds KLEE's result. We compute *the crossing time* from which on LLSPLAT *always* outperforms KLEE on the improved benchmarks from the GNU Coreutils. More concretely, for each benchmark, we analyzed a graph that describes how branch coverage evolves in 3600s using LLSPLAT and KLEE, and recorded the time when the branch coverage reported by LLSPLAT is always higher than the one reported by KLEE. Table 2.4 shows the result statistics. The crossing time of 46 and 68 benchmarks is below 60s and 120s, respectively. This fact implies that our

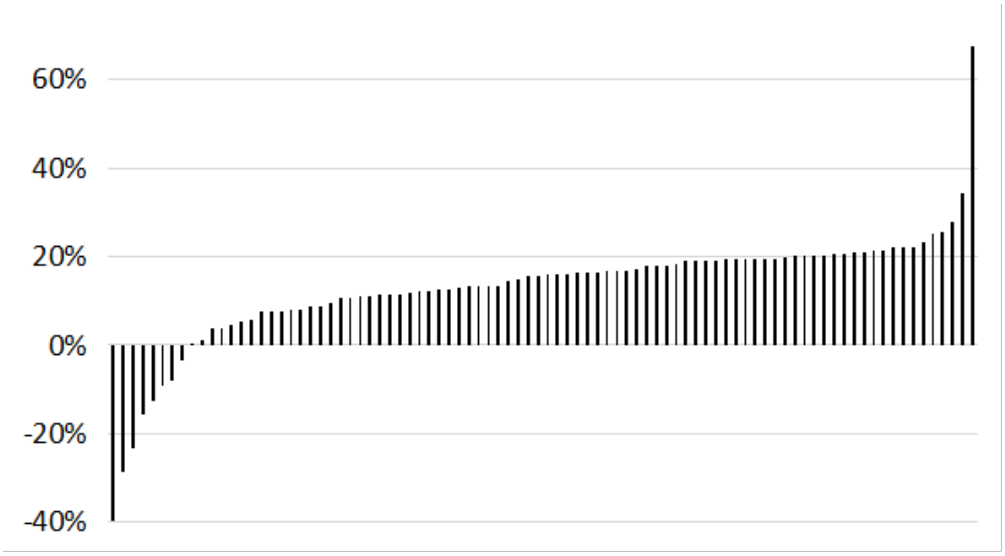


Figure 2.3: Branch coverage imporvement on the GNU Coreutils. Each bar denotes a benchmark under test. Y-axis represents relative branch coverage improvement.

Benchmark	KLEE	LLSPLAT	
	Branch covered	Branch covered	Number of governed regions
csplit	1095	955 (-12.78%)	54059
chown	883	1000 (+13.25%)	55485
shred	734	762 (+3.81%)	85194
dd	647	811 (+25.34%)	38815
cut	651	728 (+11.82%)	39396
echo	258	301 (+16.67%)	108079
uniq	686	700 (+12.24%)	44827
link	431	518 (+20.18%)	79621
nice	481	581 (+20.79%)	94622
df	844	907 (+7.46%)	81981

Table 2.3: Branch coverage comparison between LLSPLAT and KLEE and the number of encoded governed regions on 10 randomly selected benchmarks from the GNU Coreutils method is preferable over KLEE on these benchmarks given a tight time budget.

In conclusion, the concolic+BMC algorithm has a benefit of increasing branch coverage

Crossing Time(s)	0-60	60-120	120-180	>180
Number of Benchmarks	46	22	1	11

Table 2.4: Crossing time statistics of improved benchmarks on the GNU Coreutils

of all 10 programs in the Windows NT Drivers Simplified benchmark set, and 80 out of 88 programs in the GNU Coreutils benchmark set.

2.5.3 Threats to Experiment Validity

We identified the following threats to the validity of our experiment:

- **Test benchmarks used in the experiment may not be representative of all programs.** We chose the Windows NT Drivers Simplified set from [SVC] since three programs in this benchmark set are used in [SK14] to evaluate context-guided concolic testing. Thus we evaluated all programs in this benchmark set. To make our benchmark selection more unbiased, we chose 88 programs from the GNU Coreutils benchmark set used in [CDE08] to evaluate KLEE. Even though we used 98 C programs with diverse sources, they may not be representative of all programs.

- **More LLVM-IR types may yield different results.** The current implementation of LLSPLAT only deals with LLVM-IR of non-complex constant expression type and non-floating type during the BMC encoding procedure. The experimental results might be different with more other type of LLVM-IR instructions.

- **Large governed region size may yield different results.** In the GNU coreutils expriment, there are 8 benchmarks where LLSPLAT performed worse than KLEE. We observe the BMC formulas in those benchmarks are complex. Thus the STP solver used in LLSPLAT and KLEE might not be smart enough to solve them efficiently. Under such cases avoiding to encode complex governed regions or breaking complex regions into a few small regions could help improve the result. It remains as our future work to identify BMC formulas of acceptable complexity to avoid solving hard formulas.

2.6 Related Work

Concolic Testing Several approaches analyze *states* (i.e., path constraint and symbolic store) maintained by concolic testing so as to explore the search space efficiently. Godefroid [God07] introduced compositional concolic testing. The work was later expended to do compositional concolic testing on demand [AGT08]. The main idea is to generate function summaries for an analyzed function based on the path constraint, and to reuse them if the function is called again with similar arguments. Instead of computing dynamic underapproximations of summaries, we compute exact summaries of governed regions using the static representation of the CFG. Kuznetsov et al. [KKB12] introduced the dynamic state-merging (DSM) technique. DSM maintains a history queue of states. Two states may merge (depending on a separate and independent heuristic for SMT query difficulty) if they coincide in the history queue. Our concolic+BMC approach is different because we do not analyze the states to merge execution paths.

Moreover, several approaches combine other testing techniques with concolic testing together. Majumdar and Sen introduced hybrid concolic testing [MS07] that combines random testing and concolic testing. Boonstoppel et al. proposed RWSet [BCE08], a path pruning technique identifying redundant execution paths based on similarity of their live variables. Jaffar et al. [JMN13] used interpolation to subsume execution paths that are guaranteed not to hit a buggy location. Avgerino et al. [ARC14] combined static data-flow program analysis techniques with concolic testing. Santelices et al. [SH10] introduced a technique that merges multiple execution paths based on the control dependency graph of a program.

Recently there are some other trials on the combination use of concolic testing and model checking. Daca et al. [DGH15] proposed an approach combining model checking and concolic testing whose framework is similar to [MS07]. It ran concolic testing first to generate test cases. When it failed to meet certain goals, it switched to model checking to prove path feasibility to reduce searching space. Czech et al. [CJW15] used condotional model checking to construct a residual program that is fed into a concolic testing tool to reduce testing

effort. Gonzalez-de-Aledo et al. [ASH16] exploited static analysis to zoom into potential bug candidates and concolic execution to confirm these bugs. Our method is different from these methods because our model checking targets at the loop-free fragment of the code such that we can encode multiple paths on the fly to alleviate path explosion and increase branch coverage.

Several heuristic-based approaches have been proposed to guide an execution toward a specific branch. CREST [BS08] introduced four search strategies, as already shown in the experiments. SAGE [GLM12] introduced generational search that selects all the branches in an execution path and generates a set of inputs. Xie et al. [XTH09] proposed a fitness-guided search that calculates fitness values of execution paths to guide the next execution towards a specific branch. Li et al. [LSW13] proposed a subpath-guided search which steers symbolic execution to less traveled paths. Seo and Kim [SK14] introduced context-guided search that selects branches in a new context to help prevent the continuous selecting of the same branch. KLEE [CDE08] used a meta-strategy which combines several search strategies in a round robin fashion to avoid cases where one strategy gets stuck. Conceptually, our concolic+BMC approach is compatible with the above search heuristics. In essence, concolic testing is to cover both branches of a conditional, which can be regarded as two destinations. Then the goal of concolic+BMC is to cover all destinations in a program.

Bounded Model Checking VC generation approaches in modern BMC tools can be classified into two categories. The first one is based on *weakest preconditions* [Dij97] by performing a demand-driven backward analysis from the points of interest [BL05, CFS09, FS01, LQL12]. The other one encodes a program in a forward manner, such as CBMC [KCY], ESBMC [CMN12], and LLBMC [MFS12]. We are inspired by the VC generation algorithm of CBMC, and thus conceptually it is the closest work to our BMC algorithm. The VC generation of CBMC differs from ours in four ways. First, though CBMC also does variable renaming, it does it using a fixed order of basic blocks. We relax this requirement and prove that any topological order works for variable renaming. This is important to us, because we do not have to follow the fixed order CBMC uses. In fact, we use the reverse post order

of a governed region as our topological order for variable renaming because it has been computed during the construction of depth first spanning tree which identifies back edges. We save the computation time in this way. Secondly, though the VC generation of CBMC also computes edge formulas for each basic block in a given acyclic program, *all predecessors* of the basic block contribute to deriving edge formulas. However, this is not the case in ours. For example, suppose that *gov* is governor, d is a destination of $GR(gov)$, and there is a predecessor $BB \notin GR(gov)$ of d. This case may happen because BB is polluted. Then our BMC algorithm does not derive an edge formula from BB for d. Thirdly, CBMC does not have the notion of destinations. Since a governed region may have multiple destinations, it is not clear that no matter which destination is chosen, whether the final version of variables in the formula ϕ that encodes the governed region always represents the value of the variables when the destination is reached. We prove this fact in our paper. Lastly, since CBMC encodes the entire program, it does not identify *acyclic portions of a program* using the notions such as governors. It also does function inlining and loop unrolling, which we do not.

ESBMC follows the VC generation algorithm of CBMC. It extends BMC to check concurrent programs. LLBMC explicitly models the memory as a variable representing an array of bytes, which requires LLBMC to distinguish if a little-endian or big-endian architecture is analyzed. They are orthogonal to LLSPLAT.

Software Model Checking Large-block encoding [BCG09] is widely used in software model checkers. It encodes control flow edges into one formula, for computing the abstract successor during predicate abstraction. Selective enumeration using SAT solvers [HSI10] and symbolic encodings for program regions, e.g., to summarize loops [KST13], have been successfully exploited in software model checking.

2.7 Conclusion

Since concolic testing suffers from path explosion, we introduce the concolic+BMC algorithm that applies BMC locally targeting at loop-free code fragment during concolic testing to

alleviate path explosion, and thus improve branch coverage. Our experiments show that the concolic+BMC algorithm increases branch coverage of the two test benchmark sets. It is also worth mentioning that, if we notice that there is a BMC region containing a potential error after concolic+BMC, and we want to check exactly which branch in the region leads to the error, we can always set that region to do concolic execution while keeping the remaining regions performing BMC.

We believe some future work can be achieved on top of our algorithm. Specifically, the implementation of LLSPLAT performs BMC encoding if there exists an acyclic graph containing a few merging basic blocks without function calls inside of the graph. We avoid encoding function calls because it may incur exponential blowup in the BMC formula generation. We would like to come up with a clever evaluation procedure that identifies “cheap" function calls that can be encoded. In addition, solving functions of large scale may yield long time being spent in the SMT solver. We would also like to investigate whether there exists a low cost governed region overhead estimation method to make the selection of BMC regions more intelligent.

CHAPTER 3

Codecomb: Automated Test Case Generation And Defect Detection for Emebedded Software Based on Symbolic Execution

3.1 Introduction

Reliability of the embedded system has gained more and more interests with the development of the society. Especially in the field of consumer electronics, industry control, military system and aerospace, huge amount of embedded system modules are playing crucial roles to achieve various functions. Similar to conventional software development, validation & verification often spans more than 50% of embedded software development cost. For high-standard embedded systems, the total test cost is about 60% of project. In critical systems, this percentage is even greater than 80%. Thus a complete and thorough software testing technique for embedded systems is a core research topic in software testing to guarantee the reliability of software systems.

Conventional testing methods use a hybrid of manual testing and simple semi-automation tools. Those methods can not guarantee satisfied efficacy in the sense of code coverage and run time. In addition the manual part introduces great manpower. The occurrence and development of symbolic execution provides a new breakthrough for test case automatic generation. Test case generation based on symbolic execution greatly improves the coverage of code under test, and vastly reduces the redundancy of generated test cases [CS13, SPF14, Ueh16, CZG13].

There are a lot of advances in the development of symbolic execution tools in recent years.

CREST [BS08] is an open-source symbolic execution tool developed for C program. It uses CIL to format and instrument C program under test. While the program is executing, CREST collects path constraints and computes test case for a new path to execute. The disadvantage of CREST is that it lacks supports some data types such as pointers and floating points numbers. SAGE [GLM12] is a symbolic execution tool developed by Microsoft. It is reported to have found one third of the software deficiencies in the development of Windows7. But SAGE only supports x86, and does not support other embedded platforms. KLEE [CDE08] complies code under test into LLVM-IR, and then performs symbolic execution over LLVM-IR. It can generate test case with high coverage. KLEE can not symbolically execute library function calls and third-party libraries without having source code. Thus the path constraints obtained by KLEE are not always precise enough. S2E [CKC11] achieves selective symbolic execution technique based on KLEE and QEMU. For complex source codes S2E can select part of the source code to perform symbolic execution, the remaining part can be executed on QEMU. S2E improves KLEE in the sense that programs without source codes are able to run at the emulated machines. This fact makes testing on top of embedded platform possible. However, due to the limitation of the QEMU itself, S2E can only support limited embedded platforms. In addition, there are also some constraints on the accuracy of system information that S2E collects.

Some testing of embedded software are implemented on emulators. There are a few limitations comparing with testing on an emulator and the real embedded platform. First, run-time performance of an emulator is usually slower than real embedded platform. This performance difference is due to the fact that executing instruction in another instruction set such as ARM on top of a x86 machine takes extra work to handle compatibility issue. In practice, even if some emulators are in the form of X86 image such as Android emulator, many applications are not able to run on those emulators. Second, emulators are customized for certain embedded platforms, which is not universal. Even in a specific emulator such as Android emulator, due to the customization on the Android platform from different manufacturers, the emulator may not achieve the same functions as the real board.

Embedded platforms usually have limited hardware resources and have various layouts.

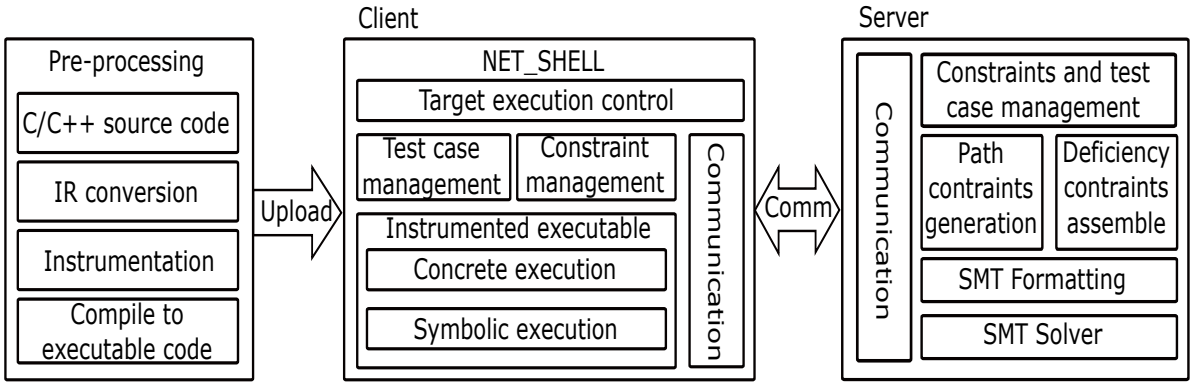


Figure 3.1: Codecomb architecture

In symbolic execution, the highest computation load is on SMT query of path conditions and property checking. Since the complexity of those SMT queries are NP, using the computation power on the embedded platform is thus not practical and ideal. However, the symbolic executor only needs the query result in order to proceed exploration. It means that the SMT query load can be transferred to other powerful machines to become independent from the path execution. In this chapter, we adopt symbolic execution and automatic test case generation into embedded platforms. We use a client/server model to separate constraint solving and symbolic execution to greatly reduce the computation load in the embedded platform. Theoretically Codecomb can be tailors to any embedded platform that has communication modules to send and receive data. We also provide symbolic descriptions for corresponding memory deficiencies. Experiments on PC and Pandaboard embedded platform show that Codecomb can run on the embedded platform, and is able to find software deficiencies automatically.

3.2 System Architecture

To accommodate the traits of the embedded platforms that they have limited hardware resources and have various layouts, we propose Codecomb, a client-server architecture tool that separates the code under test and the symbolic execution engine. The system architecture is shown in Figure 3.1. The architecture contains a pre-process module, a client module, and a server module. The pre-process module responses for perform instrumentation of input symbols and path constraints condition functions. The client module runs on real embedded

platforms. It executes instrumented and complied binary code to collect path constraints and extract symbolic description of memory deficiencies with in arrays and pointers. The server side communicates with the client module to perform SMT solving so as to obtain real test case and check for deficiencies.

3.2.1 Source Code Instrumentation

One of the key steps for symbolic execution is the source code instrumentation. The function instrumented can collect symbolic descriptions of path constraints and deficiencies such that symbolic execution and concrete execution happen at the same time.

Codecomb uses LLVM-IR as the intermediate representation of the source code. During the instrumentation process it first recognizes input variables that are needed to symbolize. Based on symbolized input variables, Codecomb inserts different function calls to various IR statements. Sorts of the statements of our interests are assignments, branch conditions, function calls, and memory deficiency related statements ($getelementptr, alloca, etc.$). Functions instrumented include symbolic execution initialization function, load function, store function, operator application function, buffer overflow checking function, pointer error checking function, and memory leak checking function, etc. Load and store functions can read and write symbolic expressions from a symbolic store, operator application function constructs symbolic expressions for unary and binary operations of the LLVM-IR. As for property checking, buffer overflow checking function collects buffer size and the current index in the buffer, pointer error checking function collects the pointer address and the space that a pointer points to, and memory leak function collects memory allocation information during execution. When a branch instruction is executed, an instrumented branch checking function collects path constraints. If the instruction is a function call, instrumentation process inserts a function call to collect call stacks. Once the instrumentation completes, a cross compilation tool compiles instrumented code into executable object code such that the code can be uploaded to the client module to perform symbolic execution.

3.2.2 Client Module

The client module runs on the target embedded board. It includes a shell program and an executable instrumented bytecode. The shell program performs target execution control, test case management, constraints management, and communication with the server module. Target execution control sub-module initializes the server module in the shell program, and makes sure the handshake with the server module is a success. Test case management sub-module sends, receives and stores the test case from the server side solver engine. Constraint management sub-module maintains all the symbolic expressions generated from the symbolic execution, and communicates to the server module via the communication sub-module.

After completion of all the initialization process in the shell program, the shell program randomly generate a test case for the instrumented code to start concrete execution. With the help of the instrumented function calls, symbolic execution and the concrete execution are executed simultaneously. Instrumented function calls collect path and deficiency constraints from the execution trace, and then send those constraints to the server side for SMT solving. The client side waits for the feedback from the server side to decide whether to continue testing. If the client receives a continue signal along with the a set of new test case, the client starts a new round of test.

3.2.3 Server Module

The server module responses for receiving SMT constraints from the server side, calling SMT solver to query satisfiability and test cases, and sending data back to the client. It contains the following sub-modules.

- **Communication sub-module**: this module receives path constraints from the client, and sends control signal and new test case back to the client. In Combcomb we adopt LAN to achieve communication between the client and the server.
- **Constraints and test case management sub-module**: this module maintains a work list of all unexplored path constraints, and manages corresponding test cases

generated by the SMT solvers. The server sends test cases in this module to the client to perform next round of execution to explore an undiscovered path.

- **Path constraints generation sub-module**: this module systematically generates new path constraints based on the path sent from the client module. In Codecomb we perform depth-first style path reasoning. All branches are inverted on by one from the end of the path to the beginning of the path so that the SMT solver can verify the reachability of the new path. A reachabable path along with the corresponding test case is sent to the constraints and test case management sub-module.

- **Deficiency constraints assemble sub-module**: this module conjoins the union of all deficiency constraints to a given path constraint.

- **SMT formatting sub-module**: this module changes the path constraints to specific format that a SMT solver can read such that the server can call different SMT solvers on demand.

- **SMT solver sub-module**: this module calls the corresponding SMT solver to query satisfiability in parallel, and return test cases.

In real implementation the server side runs in multi-threaded mode to perform path reasoning, SMT path and deficiency constraints solving. The main thread queries all symbolic constraints via SMT solvers that are run in children threads, and sends a set of new test cases from the constraints and test case management sub-module back to the client. Some children threads are responsible for checking deficiency properties as well. If a path is proved to be faulty, the error test case will be sent back to the client side to reproduce the error.

3.3 Test Case Generation

Test case generation based on symbolic execution generates a path constraint by conjoining symbolic expressions of all branch conditions of a program path. Inverting one of the clauses of the path constraint yields a new program path. Exhaustively and recursively inverting

every condition enumerates all program paths.

An example of test case generation is shown in Table 3.1. Assume that after initialization the symbol a has been assigned with a value greater than 2, then at line 5 the path constraint set is $S = \{3 * a + 2 \neq 8\}$. Thus line 6 will be executed. The server module inverts the condition in S, and query for a new path $S' = \{3 * a + 2 \neq 8\}$. Test case $a == 2$ is then sent back to the client to execute line 5 in the next round of execution. When the program

Table 3.1: Example of test case generation

Example: simple.c	
1	int main(void) {
2	int a, b;
3	mksymbol_int(a);
4	b = 3 * a + 2;
5	if (b == 8) printf ("8/n");
6	else printf ("not 8/n");
7	return 0;
8	}

under test is complex and contains multiple branches, after one round of execution the path constraint set is $S = \{s_1, s_2, ..., s_n\}$. The server side changes the last condition in S to generate a new path $S' = \{s_1, s_2, ..., s'_n\}$. s_n is labelled as visited in order not to query redundant program paths. Another new path is generated by inverting the last unvisited condition in path under analyzing of the path constraints generation sub-module. This depth-first search scheme guarantees visiting all program paths without querying repetitive constraints. Note that other path selection and generation schemes are also valid as long as they systematically enumerate all program paths.

3.4 Defect Detection

Compilers and program IDES can identify regular syntax and semantic errors. Some IDEs can achieve part of the static analysis functions. However, checking deficiencies such as memory overflow and memory leak are hard for static analysis tools. Symbolic execution can check such deficiencies thoroughly with high code coverage. We listed all the deficiencies we implemented to check in Codecomb below.

3.4.1 Buffer Checking

Conventional static analysis only compares the concrete index of an array with the size of the array to check whether a buffer overflow occurs. However, if the index is represented as an expression instead of a concrete value, it is hard for a static tool to check for the potential buffer overflow risk. Same thing happens to a pointer that points to an array. Codecomb handle checks for such risks by instrumenting function calls before indexing *getelementptr* LLVM-IR statement. Such functions can obtain symbolic and concrete value of the index, and corresponding C/C++ source code line number. A *arraycheck* function checks whether there is an array out of bound, a *pointercheck* function checks whether the current pointer is out of bound, and a *ptrtoarraycheck* function checks whether a pointer that points to an array is out of bound. Thus functions first determines whether current index symbolic expression can be greater than the buffer size to check for buffer overflow or out of bound indexing. If a violation happens an error reproduce test case along with an error is returned. For example, there is an integer array that allocates M elements. Indexing $a[i]$ returns a value in a. If i is an expression instead of a concrete value, instrumented functions adds a constraint $i \geq M$ to the deficiency constraint set R. R will be conjoined with the path constraint set S in the server side to check for violations. If conjoined constraints returns a satisfiable result, the indexing statement has the risk of buffer overflow. Using the example shown in Table 3.2, Codecomb mainly checks the following three overflow errors.

Table 3.2: Example of deficiency detection

C souce code		LLVM-IR (snippet)
int i;	1	%i = alloca i32
int s[5];	2	%s = alloca [5xi32]
int *p = s;	3	%p = alloca i32*
int *q = (int*) malloc (5*sizeof(int));	4	%q = alloca i32*
s[i] = 1;	5	%2= getelementptr inbounds[5xi32]* %s, i32 0, i32 0
p[i] = 1;	6	store i32* %2, i32** %p
q[i] = 1;	7	%3 = call noalias i8* @malloc(i64 20) #2
	8	%4 = bitcast i8* %3 to i32*
	9	store i32* %4, i32** %q
	10	%5 = load i32* %i
	11	%6 = sext i32 %5 to i64
	12	%7= getelementptr inbounds[5xi32]* %s, i32 0, i64 %6
	13	%8 = load i32* %i
	14	%9 = sext i32 %8 to i64
	15	%10 = load i32** %p
	16	%11= getelementptr inbounds[5xi32]* %10, i32 0, i64 %9
	17	%12 = load i32* %i
	18	%13 = sext i32 %12 to i64
	19	%14 = load i32** %q
	20	%15= getelementptr inbounds[5xi32]* %14, i32 0, i64 %13

Array index out of bound In Table 3.2, we define an array $ints[5]$, an indexing of $s[i]$ where $i \geq 5$ will trigger index out of bound error. The indexing generates a *getelementptr* instruction in the corresponding LLVM-IR. At line 2 of the LLVM-IR, the array of 5 integers is allocated. At line 12 of the LLVM-IR, we can obtain the indexing offset is %6, which corresponding to i in the source code. At this time, we instrument a stub function before line 12 by passing in %6, the size of s, and the current line number in the original source

code. Inside the stub function, Codecomb checks whether %6 can be out of the s's bound. A concrete value of i can be compared directly. If i is a symbolic expression associated with the input symbols, Codecomb calls the server side to check whether such a deficiency is satisfied in the current execution path. If such a violation exists, the line number of the original source code is printed.

Array pointer out of bound The indexing of the array pointer is at line 16. From line 2 of the LLVM-IR, we acknowledge that the array is of 5 integers. From stub functions inserted before line 5,6,15 we are able to know %10 actually points to the array s. Thus we insert a stub function before line 16 by passing in the bound and the current index to check whether there is a buffer overflow violation by querying the SMT solver in the server side. If such a violation exists, the line number of the original source code is printed.

Pointer segmentation We can observe from line 7 of the LLVM-IR, the pointer points to a new allocated space of 20 bytes where the pointer can be dereferenced to a 32-bit integer data. The indexing of the array pointer is at line 20. We insert a stub function by passing in the index offset %13, the total allocated space of the array pointer, and dereferenced data size. Inside the stub function, Codecomb checks whether %6 can be out of the s's bound. A concrete value of i can be compared directly. If i is a symbolic expression associated with the input symbols, Codecomb calls the server side to check whether such a deficiency is satisfied in the current execution path. If such a violation exists, the line number of the original source code is printed.

3.4.2 Memory Allocation and deallocation

In symbolic execution, we can insert stub functions(malloccheck, freecheck, mallocinit) to obtain all the information of the memory operation within an execution path so as to simulate the usage of the memory. At the end of an execution path, Codecomb checks whether all allocated memory spaces have been deallocated. If such a violation exists, the line number of the original source code is printed. Checking for double-free violation is similar to memory

leak checking. In a program path, if the stub function $freecheck$ detects that the program is freeing up a space that has already been deallocated, Codecomb returns a double free error along with the corresponding line number. Detailed implementation is shown as follows.

Memory leak detection Codecomb maintains a memory usage table. The corresponding stub function updates the usage table by adding the allocated memory address and size, original source code line number, and a flag of usage. Value of 1 in the flag means the memory has not been deallocated. While the program executes a free instruction, the stub function queries the memory usage table. If there is an corresponding entry in the table, the flag value is decremented by 1. At the end of the execution, if there exists an entry in the memory usage table that has a flag value equals to 1, Codecomb returns memory leak error with corresponding source code line number.

Double free detection Double free error often happens on large scale and complex software systems. With the help of the memory usage table, double free detection is available by checking the flag value in the corresponding entry in the table. A flag value less than 0 means that the program is trying to deallocate a space that has already been released. When the program executes a memory release instruction, a stub function is called to query the memory usage table. If such a error is detected in the stub function, Codecomb returns memory leak error with corresponding source code line number.

3.5 Experimental Results

We have built our Codecomb prototype that implements the automatic test case generation for embedded pplatforms. We implements the server on top of a Ubuntu 14.04 computer. In the server side we use Z3 SMT solver as the symbolic execution querying solver. For the comparison purpose, the client is implemented on Pandaboard, an embedded development board that runs Ubuntu 12.4, and on a Ubuntu 14.04 computer. Using a linux development board we can cross compile Valgrind to show the effectiveness of Codecomb. Thus we cross

compiled Valgrind 3.11.0 on the Pandaboard. We list one of the test cases as below to clearly observe the performance, and the reports from CodeComb and Valgrind are shown in Table 3.3.

```
1  int a[5] = {1,2,3,4,5};
2  int *p;
3  p = a;
4  int *q = (int *)malloc(5*sizeof(int));
5  int *q2 = (int *)malloc(5*sizeof(int));
6  int *q3 = (int *)malloc(5*sizeof(int));
7  int i,j;
8  mksymbol_int(i);
9  mksymbol_int(j);
10 if (i < 10) {
11   a[i] = 0;
12   if (j < 8) {
13     q[j] = 1;
14     p[j] = 1;
15     p[j] = a[i] + 1;
16   } else {
17     free(q3);
18   }
19 }
20 free(q2);
21 free(q2);
```

We observe that both Codecomb and Valgrind report all potential errors in the test case. However, to trigger all errors, Valgrind requires carefully selected test cases. For complex source code, it is hard to find all test cases that can trigger all errors manually. Codecomb generates all test cases automatically, and can finish the whole flow in a single run. There is no need to design special test cases comparing with Valgrind. We show the comparison

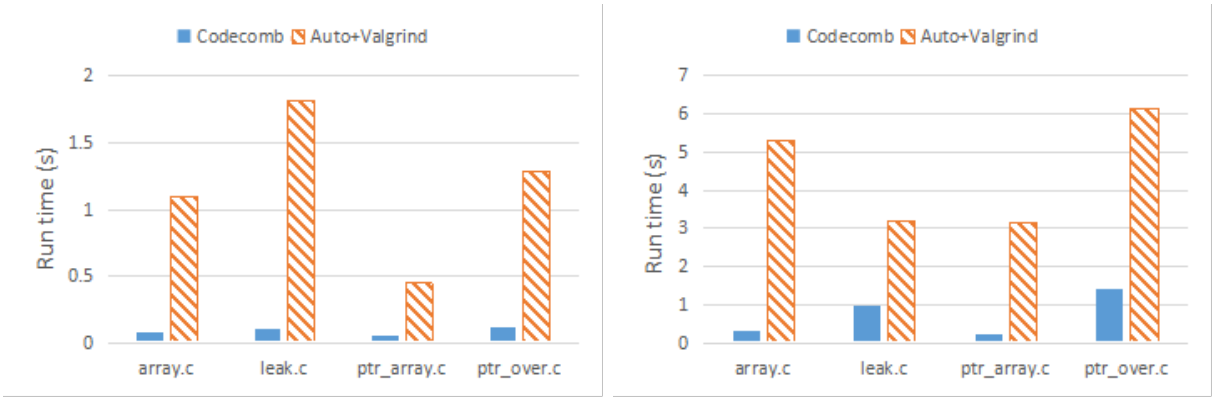
Table 3.3: Reports of CodeComb and Valgrind for bugtest

Codecomb Report	Valgrind Report
Branch coverage 100%	Test1:(i = 0, j = 0)
Array out of bound #11, #15	Memory leak #4, #6
Pointer out of bound #13	Pointer double free #21
Array pointer out of bound #14, #15	Test2:(i = 11, j = 11)
Memory leak #4, #6	Memory leak #4, #6
Pointer double free #21	Pointer double free #21
	Test3:(i = 4, j = 10)
	Memory leak #4
	Pointer double free #21
	Test4: (i = 4, j = 3)
	Memory leak #4, #6
	Pointer double free #21
	Test5: (i = 9, j = 7)
	segmentation fault #13, #14, #15

results of other test cases in Table 3.4. In this table, we provide three testing schemes. The first scheme (Random + Valgrind) uses Valgrind by feeding in random inputs for multiple rounds. The second scheme (Auto + Valgrind) uses Valgrind by feeding in all test cases generated from Codecomb. The last scheme (Codecomb) uses pure CodeComb to conduct testing. From the experiments we observe that Codecomb can detect all the test cases that Valgrind can detect automatically. Due to the fact that Valgrind actually execute one single program path determined by the input, carefully selection of the test cases are required to find all potential errors. Thus the random testing scheme using Valgrind can not guarantee the path coverage of the program, and can not guarantee to find all potential errors. The characteristic of automatic test case generation benefits Valgrind in the second scheme to find all potential errors so that Valgrind can visit all program paths. For all program paths,

Table 3.4: Testing result comparison between Codecomb and Valgrind. A stands for buffer overflow, B stands for memory leak, C stands for pointer double free.

Benchmark	array			leak			ptr_arr			ptr_over		
Error Type	A	B	C	A	B	C	A	B	C	A	B	C
Random + Valgrind	0	0	0	2	1	0	0	0	0	1	1	0
Auto + Valgrind	1	0	0	2	1	0	1	0	0	2	1	0
Codecomb	1	0	0	2	1	0	1	0	0	2	1	0



(a) PC client

(b) Pandaboard client

Figure 3.2: Run time comparison between CodeComb and Auto+Valgrind

Codecomb detects the same vulnerabilities in the program as in Valgrind, and Codecomb shows the better automation than Valgrind.

We also compare the deficiency checking speed between Codecomb and Valgrind, and the result is shown in Figure 3.2. Figure 3.2a shows the run time comparison running on the PC client, and Figure 3.2b is the comparison on the Pandaboard client. The performances on the Pandaboard are in general worse than the PC client because embedded platform has limited hardware resources and computation power. We profiled the performance of CodeComb, and found the communication takes great amount of the total run time. Even though network communication is a huge overhead in Codecomb, we can still observe that Codecomb itself has better performance than Valgrind using test cases automatically generated from Codecomb on both settings.

In conclusion, in this experiment we preliminary show that Codecomb can perform symbolic execution and automatic test case generation on embedded system platforms. Under the Linux embedded platform settings we show that Codecomb can detect the same memory deficiencies as in Valgrind. Comparing to Valgrind that needs to carefully select test cases, Codecomb is a fully automatic approach on test case generation on embedded platforms. As Codecomb is an executable program, theoretically it has the ability to run on non-Linux embedded platform where Valgrind can not be installed as long as the cross compilation tool and network communication port on the client are provided.

3.6 Conclusion

In this chapter, we adopt symbolic execution and automatic test case generation into embedded platforms. We use a client/server model to separate constraint solving and symbolic execution to greatly reduce the computation load in the embedded platform. Multi-threading in both client and server modules increases the efficiency of the Codecomb. Experiments on PC and Pandaboard embedded platform show the Codecomb can run on the embedded platform, and is able to find software deficiencies automatically. As a future work, we will continue improve Codecomb to make it suitable for more complex source code, and add support to more communication ports to make Codecomb suits more sorts of embedded platforms.

CHAPTER 4

A Dynamic Approach to Functional Verification of High Level Synthesis

4.1 Introduction

High-level design has many advantages over the commonplace design flow that begins with register-transfer level (RTL) code. Among the most compelling advantages is the improved verification efficiency which a higher level of abstraction offers. It is apparent to the point of being self-evident that when the source code of a design is created, there will be fewer errors if the source is at a higher abstraction level than if it is at a lower level. However, there is still a process to verify the transformations which are applied to the design description as it proceeds through the design flow from creation to final realization.

Formal verification proves the correctness of the system with respect to a certain formal specification or property. For example, many testing tools exploit formal verification in bug finding [GHM16] [GWY17], and probabilistic model checking [GHW17] [GWH18] is applied to verify its reliability and energy trading behaviour. Among all formal verification categories, functional verification is a useful functionality that can be applied to HLS in a similar manner to its application in RTL-to-gate equivalence checking. Most of functional verification work with formal verification above rely on static analysis on both behaviour and generated RTL side, where model checking is widely used. A. Mathur *et al.* [MFC09] introduced the model checking on system level models and sequential equivalence checking in the HLS level functional verification. slong this line C. Marquez *et al.* [MSC13] proposed a Sequential equivalence checking (SEC) formalism and an algorithm, for use between a specification written at electronic system level, and an RTL implementation. T. Nishihara *et*

al. [NMF06] presented a new word-level equivalence checking method between two models before and after HLS or behavioral optimization. S.Kundu *et al.* [KLG08] proposed a HLS verification algorithm defined by equivalent bisimulation relation. A. Koelbl *et al.* [KJJ09] discussed solver technology that has shown to be effective on many real-life equivalence checking problems. It compared different solver strategies, and presented a hierarchical checking methodology. In HLS verification field, many existing HLS functional verification work focus on scheduling verification only. C. Karfa *et al.* [KSM08] described a formal method for checking equivalence between a given behavioral specification prior to scheduling and the one produced by the scheduler with operations on finite state machine with datapath (FSMD) models applied to both the behaviors. C. Lee *et al.* [LSH11] utilized FSMD on both descriptions as well, and their goal is to verify scheduling with code transformations such as speculation and common sub-expression extraction across basic block boundaries. J. Urdahl *et al.* [USK14] proposed a path predicate abstraction for sound system level models of RTL system-level designs model checking based on the notion of operational graph coloring. One issue for the above static analysis methods is that static analysis tries to prove the general equivalence of two state transition machines with full automation, which is hard in general. Another issue is that since model checking describes all the state transitions of a program model in their formulas including non-critical data paths and components that may not affect the output, the number of states grows exponentially along with the increasing scale.

Comparing with static analysis, dynamic testing method such as symbolic execution and concolic execution explores model under investigation on a path-by-path basis. A single path has much lighter encoding than a whole model. It can summarize both combinational and sequential circuit trace accurately. In practical situation, a circuit can generate output within an estimable clock bound, which is actually a limited part of the whole model. Path based exploration then has the advantage of easily checking all variable state transitions through all paths. It also avoids loop unrolling problem since it systematically enumerates all paths. Along this line, R. Mukherjee *et al.* [MTK16] developed a tool V2C that translates Verilog to C such that dynamic testing for sequential programs can be applied on Verilog. V2C accepts synthesizable Verilog as input and generates a word-level C program as an output.

Equivalence checking is then achievable on C level with the help of either static analyzing tools or dynamic execution tools. One major disadvantage of V2C in HLS equivalence checking is that this process needs to verify or to prove the transformation using V2C is equivalent to original Verilog, which is actually a reversed process of HLS functional checking. A formal equivalence proof is needed to show the correctness of the flow. Also current V2C implementation have difficulties in dealing with dependency of inter-modular combinatorial paths or combinatorial loops. It is hard to determine stability condition for large circuits to obtain an equivalent C program automatically. On the contrary, symbolic execution on top of Verilog directly does not have similar issues as it generates symbolic expressions of the register-transfer relationship precisely. Thus directly applying V2C based framework is not a natural solution for HLS functional verification. As for standalone dynamic execution for Verilog, X. Qin *et al.* [QM14] proposed a concolic execution framework for Verilog similar to [SMA05]. It first instruments the Verilog design, then interleaves concrete and symbolic simulation to obtain execution traces, and then rearranges trace to generate path constraints and concrete testing case. Different from [SMA05], however, this method summarizes symbolic path constraints using the result from concrete simulation, while concolic testing for software has symbolic expression and concrete value generated at the same time. The performance bottleneck of this method lies on the concrete simulator. In fact, as long as here are no constraints unsolvable by the SMT sovler, pure symbolic execution without any concrete simulation can achieve the same functionally. In addition, from HLS equivalence checking perspective, there are some room for optimization in reducing the complexity of SMT solver. Thus far, a full flow of symbolic methods for HLS functional verification is still missing.

In this chapter, we propose a pure dynamic HLS functional verification flow via symbolic execution using LegUp [CCA13] as the HLS engine. More precisely, our verification flow first runs light-weight symbolic execution on both C code and HLS generated Verilog code to generate SMT execution trace, respectively. Our on-the-fly symbolic execution on Verilog side is implemented on Verilog's SMT representation obtained by Yosys [Wol] synthesis tool. Then we perform optimization on module input symbolization and valid datapath identification

from the HLS code generation perspective to greatly reduce the complexity of the SMT solver. Specifically, signals in Verilog traces are collapsed from clock-based encoding to clock-invariant version-based encoding. Considering HLS binds the same operation to an identical set of hardware, we perform one-on-one trace matching by abstracting identical operations on C and Verilog traces such that the scalability issue can be alleviated. Various experiment results show that our framework has the ability to verify the functional equivalence for numerical computing circuits, and prove the potential of using formal methods for practical HLS functional verification flow for future.

The contributions of this paper are summarized as follows.

- We propose a pure symbolic execution flow to tackle functional verification of HLS. To the best of our knowledge, it is the first time that a pure dynamic approach is applied on HLS functional verification without transforming one side of the source code to the format of its counterpart before verification.

- We present a Verilog pure symbolic execution framework without performing concrete simulation that runs directly from Verilog's SMT representation. The framework considers that multiple instructions are executed in parallel within a single clock cycle, and generates symbolic expression on-the-fly without the help of concrete simulation. We also perform optimization on module input symbolization from HLS code generation perspective and clock-invariant version-based encoding to greatly reduce the complexity for SMT solver.

- We introduce an operation abstraction method to alleviate the scalability issue considering allocated hardware usage from HLS along the data path.

The rest of the paper is organized as follows. Section 2 reviews HLS and symbolic/concolic execution. Section 3 describes our HLS functional verification framework. Section 4 presents experimental results, and Section 5 concludes the paper.

4.2 Preliminary

4.2.1 High-level Synthesis

HLS of VLSI system [CLN11] [MS09] is effective for hardware design. HLS converts a high-level description of a design into a RTL netlist by considering real design constraint such as area constraints and delay constraints. HLS parses high-level description into an intermediate representation called control data flow graph (CDFG) [CZ06]. A CDFG $G(V_c \cup V_d, E_c \cup E_d)$ combines the control flow graph $G(V_c, E_c)$ and the data flow graph $G(V_d, E_d)$ of a design. Data operations are encapsulated into many basic blocks, and these basic blocks branches to other basic blocks representing control relationship.

All Operations in a CDFG are rearranged and transformed by scheduling, allocation and binding process so as to be mapped into an RTL netlist. Allocations determines how many resources are required in the hardware module. It tries to enable concurrency among operations to allow hardware resource sharing. The degree of concurrency is determined by the scheduling process. For the scheduling process, the results of a scheduler can be represented by an finite state machine (FSM) style state transition graph (STG) transformed from a CDFG. A STG $G(s_0, S, E)$ is a directed graph containing a starting state s_0 and a set of control state basic blocks S where each control state $s \in S$ contains a set of guarded operations that are scheduled in s. E represents for the state transition in between each state basic block s. The scheduler constructs an STG Gs such that every operation is assigned to at least one state in Gs, and all constraints are satisfied. In the meantime, the final latency in a particular performance measure is minimized. Binding assigns each scheduled operation to one of the available hardware resources. For example, if there are two add operations in the same state s of FSM after scheduling for that part, they must be assigned to two different adder resources. Scheduling and binding can be done separately or simultaneously. Since a lot of transformations are performed in the HLS to generate a RTL netlist, functional equivalence checking between behaviour description and RTL description is needed.

4.2.2 Sysmbolic/Concolic Execution For Sequential Programs

Symbolic/Concolic execution systematically explores all feasible paths in a program via actual execution of the program. The key characteristic of symbolic/concolic execution is that it uses symbolized and concrete variable representations such that all the variables defined in the program are symbolic expressions during execution. To be specific, external input variables are labelled as symbols. All left values of instructions inside a program path are encoded into symbolic expressions derived from input symbols. Expressions on branch conditions along the path constitutes a path constraint. SMT solver plays a important role in symbolic/concolic execution. It reads in constraint expressions constructed from the path, and returns the satisfiability of those constraints, which determines whether forking a new path is needed. For example, In branch instructions, symbolic/concolic execution queries a SMT solver whether its true/false branch is reachable or not. A new program path is available to explore if both branches are reachable. As a further application, property checking can be done by conjoining property expressions into path constraints. A set of concrete inputs that follow violate the property can be obtained by feeding corresponding expressions into the SMT solver, and obtaining a concrete model.

To begin with the detailed procedure, symbolic/concolic execution first instruments a program P, outputs an instrumented program P'. In P', all external inputs or user-specified execution parameters are replaced with symbols. A program executor ex is initialized when P' is ready. ex is an interpreter that responses for symbolically executing the P' and maintaining all the execution states s_i. An execution state s stores all information of the program execution such as current instruction counter, path constraints, memory storage, local registers, function call paths, and coverage statistics. , which is essentially the frontier of a program path under execution. ex maintains an execution state working list *states* storing all s_i that are able and going to visit. Once *stats* is initialized, ex first fetches one of s_i from *states*. There are various of state selection algorithms available. A few common selection scheme including depth-first path searching, bread-first path searching, process tree based random path searching, and red-black tree based random weight searching. The selected s_i is

then executed by ex. Changes are made on s_i depending on the type of instructions. For non-branch instructions ex increments $state$'s instruction counter, updates its local register value symbolically and concretely, and performs read/write operations on its memory if needed. If the instruction is a branch instruction containing a branch condition expression $cond$, ex queries a SMT solver to check whether two potential branches are possible to continue exploring. Specifically, ex queries the solver if $currPath \wedge cond$ and $currPath \wedge \neg cond$ are both satisfiable. Concrete execution plugs in concrete values to go through hard-to-solve constraints. If both queries are satisfiable, ex forks a new execution state s_j, then adds $cond$ to s_i's path constraint. Correspondingly s_j adds $\neg cond$ to its path constraint. Both states are then pushed back to the $states$ to conclude the current execution cycle. If only one path is satisfiable, there will be no fork in s_i. Unreachablility of both branches invalidates s_i.

4.3 Verification Flow

Our verification flow enumerates all possible executing paths from both C description and generated Verilog. Then it compares input/output equivalence of those paths. The main flow of the HLS functional verification is shown in Fig. 4.1. To begin with, a C behaviour program

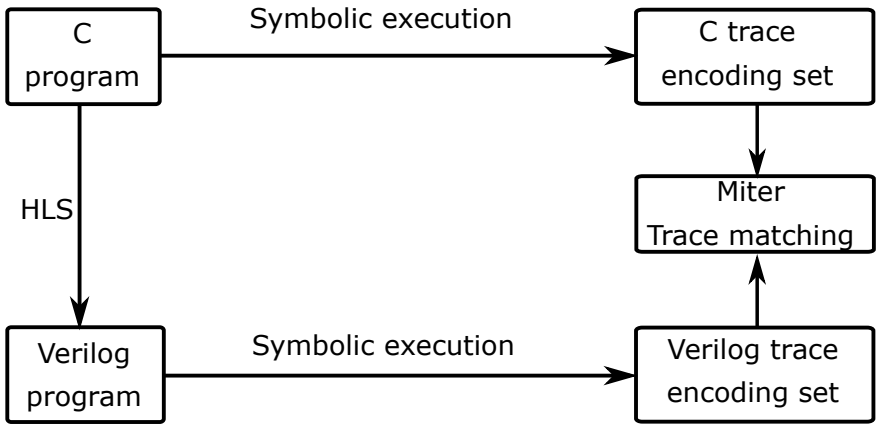


Figure 4.1: High-level synthesis functional verification flow

dut_C performs concolic execution implemented on top of KLEE [CDE08]. Different from property checking querying property along each path, we modified KLEE to conjoin output symbolic expressions to the path constraints. All conjoined program paths constitute an

output SMT expression set $pSet_C$. dut_C goes through standard HLS procedure to produce a corresponding Verilog file under test dut_V. To collect Verilog execution traces, we generate a SMT description of dut_V, and perform symbolic execution on top of it. Similar to C, output symbolic expressions are conjoined to each Verilog execution traces, and those traces form an output expression set $pSet_V$. The final matching process checks the equivalence of dut_C and HLS-generated Verilog dut_V. For any expression $p_C \in pSet_C$, there should be a $p_V \in pSet_V$ such that given the same input, output value $out_C \oplus out_V == 0$. We regard dut_C and dut_V are functional equivalent if all p_Cs are able to find their matching p_V.

4.3.1 Verilog Trace Collection

Appending output expressions to path constraints in C is a trivial process from a existing symbolic execution tool, we will mainly focus on the Verilog side in this section. Once we have the Verilog file, we would like to know at which clock cycle the circuit will output the result. Tradition concrete simulation reads in concrete data to obtain a specific circuit trace. Here we propose a light-weight pure symbolic execution method for HLS generated Verilog to exhaustively explore all the circuit traces. Our pure Verilog symbolic execution framework runs directly on Verilog's SMT representation. It considers that multiple instructions are executed in parallel within a single clock cycle, neglects unrelated modules and performs time-invariant version-based encoding on-the-fly.

4.3.1.1 Verilog SMT Representation

Converting Verilog to its corresponding SMT representation has been widely investigated in hardware model checking [CKY03]. SMT can be regarded as an extended version of boolean satisfiability problem where binary symbols are replaced with predicates over a set of non-binary symbols. Our flow utilizes the open synthesis tool Yosys [Wol] to generate SMT description for the Verilog circuit. It is generated from Yosys's RTL intermediate representation (RTLIR). A template of the generated Verilog module SMT formulas is shown in Fig 4.2. Since there is a one-on-one mapping between C function and Verilog module, all

the function prefixes in the formula correspond to behaviour C function names and Verilog module names.

```
(declare-sort |main_s| 0)
(declare-fun |main#0| (|main_s|) Bool) ; \clk
(define-fun |main_n clk| ((state |main_s|)) Bool (|main#0| state))
...
(declare-fun |main#12| (|main_s|) (_ BitVec 32)) ; \return_val
(declare-fun |main#13| (|main_s|) Bool) ; \reset
(define-fun |main_n reset| ((state |main_s|)) Bool (|main#13| state))
(declare-fun |main#14| (|main_s|) Bool) ; \start
(define-fun |main_n start| ((state |main_s|)) Bool (|main#14| state))
(declare-fun |main_h fct_inst| (|main_s|) |fct_s|)
...
(define-fun |main_h| ((state |main_s|)) Bool (and
(= (|main#13| state) (|fct_n reset| (|main_h fct_inst| state)))
(= (|main#14| state) (|fct_n start| (|main_h fct_inst| state)))
...
))
(define-fun |main_t| ((state |main_s|) (next_state |main_s|)) Bool (and
(|fct_t| (|main_h fct_inst| state) (|main_h fct_inst| next_state))
(= (|main#10| next_state) (|main#6| state))
...
))
```

Figure 4.2: A Verilog SMT description

The module S_{mod} consists of a set of function declarations and definitions. Declarations only declare function signatures while definitions define relationships between input parameters and output register value. Both declarations and definitions use state variable **state** and **next_state** as function parameters. S_{mod} divides into a register-transfer block and a parameter-setting block. The register-transfer block defines module's input RTL signals (wires and registers) as bit-vectors according to their own width, and declares a function to access the definition. It also declares and defines functions describing internal RTL signal values according to statements in RTLIR. In the parameter-setting block, all RTL signals' initial value, inter-module signal connections and signal state transfer relationship are defined with a boolean return value. For example, $main\_h$ connects wires in between S_{mod}'s sub-

modules. *main_t* sets the register value change from **state** to **next_state** in S_{mod}. We also modified Yosys so as to specify all registers' initial value in S_{mod}. To make symbolic execution update valid register value and advance the clock, at each clock cycle all function definitions in the parameter setting block are required to be evaluated to true.

4.3.1.2 Verilog Symbolic Execution

Without loss of generality, we assume C description for HLS is wrapped under the function *main*. A HLS-genrated Verilog file contains two sets of modules. One set of modules are converted from C descriptions. Semantically each module in this set corresponds to one function defined in C. Another set of modules are memory modules that implement memory operations governed by a memory controller module. The memory controller module connects to all other C-converted modules, and responses for management of all memory modules that implement data read/write operations in the hardware level. Thus in a Verilog generated by HLS tools, the top module contains a memory controller module instance in addition to a main module instance.

In symbolic execution for sequential programs, especially in the intermediate representation (IR) level, such memory operations are abstracted by memory instructions such as *load*, *store*, *getElementPtr* instead of a set of detailed function implementations. *ex* fetches symbolic expressions of a variable from its current execution state s_i's memory store, which can be constructed as a look-up table or a balanced tree in software level. In the scenario of Verilog, performing symbolic execution on those hardware level memory operations as the same level of C descriptions introduces extremely heavy load in SMT querying. However, at a specific clock cycle that a module needs data from memory, its input data pins can create or load the corresponding symbol from the current execution state. Thus neither concrete nor symbolic values fetched from those memories are at our interest. We then have the following claim for optimization in Verilog symbolic execution.

Claim 1. *Memory modules in HLS generated Verilog file is out of the symbolic execution domain, and the functionality of memory modules can be check separately.*

As a result, HLS generated module uses *main* instance as the root of the hierarchy to greatly reduce the complexity in the SMT solving process in Verilog symbolic execution.

Algorithm 5: Verilog symbolic execution

```
void run():
    preprocessing()
    while !workList.empty() do
        S = getExecutionStatesAtClk(curClk)
        for s ∈ S do
            executeState(s)
        incClk()
void executeState(s):
    for module ∈ modules do
        for statement ∈ statements do
            executeStatement(statement)
    curWorkList = {s}
    for module ∈ modules do
        for statement ∈ statements do
            forkNewState(statement, curWorkList)
    updateWorkList(curWorkList)
StatePair fork(s, c):
    if !validTrueBr(s.p, c)∧!validFalseBr(s.p, c) then
        return ∅
    else if validTrueBr(s.p, c) ∧ validFalseBr(s.p, c) then
        s_new = forkExecutionState(s)
        s.addCond(c)
        s_new.addCond(¬c)
        return {s, s_new}
    else if validTrueBr(s.p, c) then
        s.addCond(c)
        return {s, null}
    else
        s.addCond(¬c)
        return {null, s}
```

The overall flow Verilog symbolic execution is shown in Algorithm 5. Executor ex maintains a work list $workList$ of all possible circuit execution states. An execute state s_i contains a key-value pair symbolic store of variable name and its SMT expression, and a list of path constraint. New states are generated at each clock cycle while ex is executing ternary

ite expressions. The *run* method wraps up the procedure in high-level. The Verilog SMT description for ex to execute is the conjunction of all function definitions in the parameter setting block. Verilog SMT descriptions are preprocessed such that all inter-module and intra-module dependencies are organized in a chronological and topological order. Chronological order guarantees that values from previous clock cycle are assigned before any update at current clock clk_i, and topological order guarantees the data dependency of the data path. At clk_i, ex fetches a set of states S from $workList$ that are available to execute at clk_i, and symbolically executes all states in S.

The key difference between a sequential program and a Verilog is that all the instructions are executed in parallel. Thus different from sequential programs, state execution in Verilog is a two-fold process. First symbolic store in s_i appends symbolic expressions of all variables described in the Verilog's SMT function definition at clk_i. Unique symbolic names for variables are assigned to distinguish their value change at different clocks. Then all statements are iterated to check potential state forking *ite* statements. As stated in $fork$ method, the forking procedure reads path constraints from s_i and the condition from *ite* expression, forks a new state s_j if both the condition and its negation are reachable along the current path constraint. Forked states are temporary buffered in $curWorkList$ such that SMT solvers can query subsequent branches over all states in $curWorkList$.

Ideally the trace collected in symbolic execution can be used to match with C-generated trace directly. One major drawback of this representation is that it is a clock based representation. The solving complexity grows exponentially as the clock increases, and a typical circuit runs more than 1000 clock cycles to get an output. By taking a closer look at the register value curves using clock as horizontal axis, we notice piece-wise constant patterns for all registers. This fact is reasonable in the sense that in HLS-generated Verilog a signal will not be updated until all of its guard conditions are true. We then only need to create expressions at the clock cycle when a signal has been updated. Thus we are able to fold the trace representation from time-based to version-based to make it compact. A condition free variable in Verilog updates its version for each clock cycle because it gets assigned without constraints. For a guarded variable, it may contain multiple guards, and at clk_i there is at

most one path among those guard conditions is satisfied. In symbolic execution, for a specific branch of a path we know which branches are satisfied. The signal will be updated by a new version in a single static assignment manner if its symbolic value has been changed.

Example Consider a HLS-generated Verilog snippet performing accumulative addition of a two-element array shown below.

```
always@(posedge clk) begin
  if ((cur_state == F_BB__4_5))) begin
    result0_reg <= result0;
  end
  ...
  if ((cur_state == F_BB__8_6)) begin
    ret_val <= result0_reg;
  end
end
```

Symbolic execution claims the finish flag is set at the 14th clock cycle. The version table for corresponding RTL signals from 5th clock cycle to 14th clock cycle is shown in Table 4.1.

Table 4.1: RTL signal version for the example

Signal	Clock cycle #									
Name	5	6	7	8	9	10	11	12	13	14
ret_val	0	0	0	0	0	0	0	0	0	1
result0_reg	0	0	1	1	1	1	1	2	2	2
result0	0	1	1	1	1	1	2	2	2	2

Constraints assembled from Table 4.1 after symbolic execution are:

$ret\_val\_1 == result0\_reg\_2 \wedge result0\_reg\_2 == result0\_2$

$\wedge result0\_reg\_1 == result0\_1$

4.3.2 C and Verilog Trace Matching

After we collect C-generated traces and Verilog encoded traces, we perform one-on-one matching on traces to check equivalence. Conventional miter method performs exclusive disjunction of C and Verilog's outputs after conjoining two symbolic traces and equalizing inputs of C and Verilog. In practice, many HLS use cases are in numerical computing. In such use cases traces go through long and complex data path. For example, two operands of the multiplication can be 64-bit SMT bit-vector expressions with more than 100 clauses. The computation load for a SMT solver to evaluate the functional equivalence of many such expressions is extremely heavy, and the computation complexity grows exponentially. From HLS perspective, HLS binds multiple operations to given fixed hardware resources. For example, many multiplications in C code will be assigned to the same multiplication hardware block. Thus we can alleviate the load by abstracting such operations so that the length of two traces are reduced. As long as we guarantee the equivalence of the corresponding operands in C and Verilog before the operation, we are able to use equalized abstracted symbols to replace those operands in C and Verilog's traces during matching. Thus we propose an operation abstraction method for both C and Verilog symbolic traces that replace verified operations with abstracted symbols.

We define $traceC$ and $traceV$ by SMT expressions for C and Verilog that represent the data flow graph from input to output. We define $DFGC$ by the data flow graph of $traceC$, and $DFGV$ by the data flow graph $traceV$, respectively. Note that both $DFGC$ and $DFGV$ are directed acyclic graph (DAG). Without loss of generality, we assume there is only one single output. A super sink node that points from all the output nodes can be appended under the multiple outputs scenario. Different DFGs within a set share many sub-traces in common. If we can identify a target operation in $DFGC_i$ has a equivalent operation in $DFGV_i$, we abstract such operation as symbols. Results of SMT representation of identical operations in both C and Verilog counterpart can be replaced by an abstracted symbol as long as all the incoming operands in C and Verilog are equivalent symbolically.

The algorithm of the trace comparison method is shown in Algorithm 6. In high level, a

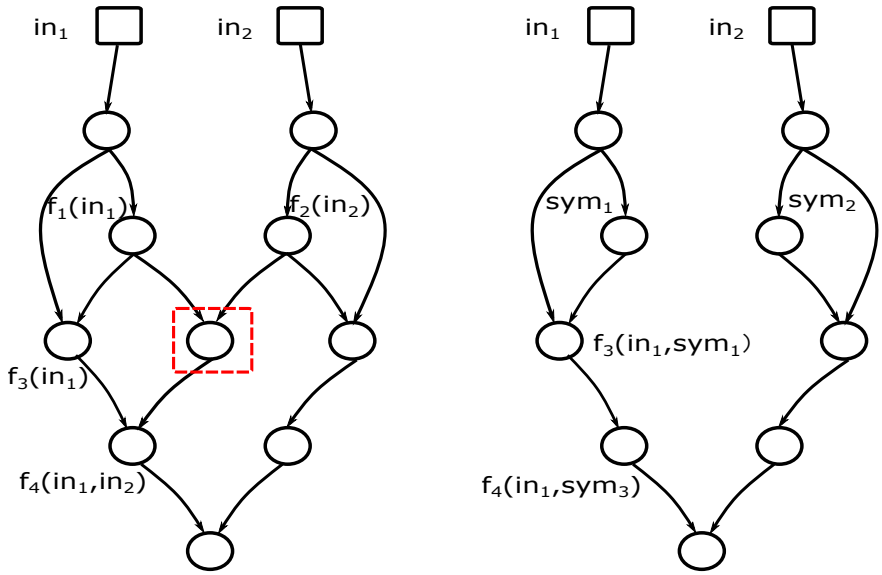


Figure 4.3: An example of operation abstraction

pair of two trace candidates *traceC* and *traceV* first match corresponding input symbols in method *connectInput*. Then the algorithm abstracts all identical operations that can achieve functional equivalence in *traceC* and *traceV*. Abstracted *traceC* and *traceV* then conjoin with a exclusive-or expression of C's and Verilog's outputs to become the final expression for the SMT solver query. An unsatisfiable result means that *traceC* and *traceV* are functional equivalent because the result from the exclusive-or of outputs has to be false. The abstraction step reduces the size of both traces, and thus make the final query step less complex.

The abstraction method first performs topological sort on both *traceC* and *traceV*, making all operations order by their data dependencies. Then it iterates through every operation *traceV*, and finds a same operation that is functional equivalent in corresponding C counterpart. The checking process iterates through *traceC*. Subtraces from the beginning till this operation on both sides are calculated. The algorithm queries the SMT solver to check whether all the operands for this operation are equivalent. If there exists such an operation, the algorithm creates new symbols to abstract the operation, and replaces all usages of this operation with the newly created symbol. *replaceSymbol* method creates new symbols for both operands and output left value. Equal relationships between symbols created in both C and Verilog are established as well. Newly created symbols then replace all the occurrences of their originally represented operations. Since operands of the operation

Algorithm 6: Operation abstraction

```
bool compare(traceC, traceV):
    connectInput(traceC, traceV)
    abstract(traceC, traceV)
    xorOutput(traceC, traceV)
    return ¬query(traceC, traceV)
void abstract(traceC, traceV):
    listC = topoSort(traceC)
    listV = topoSort(traceV)
    for clauseV ∈ listV do
        if isTargetOperation(clauseV, listC) then
            clauseC = getOperationInC(listC)
            replaceSymbol(clauseC, traceC, clauseV, traceV)
bool isTargetOperation(clauseV, listC):
    for clauseC ∈ listC do
        if op(clauseC) == op(clauseV) then
            for operandC, operandV ∈ clauseC, clauseV do
                subTraceC = getSubTrace(operandC)
                subTraceV = getSubTrace(operandV)
                if ¬euiqvalent(subTraceC, subTraceV) then
                    Continue
                return true;
    return false;
void replaceSymbol(clauseC, traceC, clauseV, traceV):
    createSymbols(clauseC, traceC, false)
    createSymbols(clauseV, traceV, false)
    eraseFromDFG(cluaseC, traceC, clauseV, traceV)
void createSymbols(clause, trace):
    for input ∈ clause do
        symName = newSymName(input)
        sym = newSym(symName)
        for use ∈ uses(input) do
            updateInputSym(sym)
    symName = newSymName(output)
    sym = newSym(symName)
    for use ∈ uses(output) do
        updateInputSym(sym)
```

may have other usages, all usages of the operands can be replaced with the newly created symbol. The accomplishment of symbol replacement means that the equivalent operation is independent from the trace. The algorithm thus removes the operation from the trace.

An example is shown in Figure 4.3, in this DFG the red dashed block is the operation with expensive verification cost, and has an equivalent counterpart in converted code. To erase this node from the DFG, we create symbols for all data of the incoming and outgoing edges, and update all related nodes with the created symbols.

4.3.3 Termination of The Flow

Theoretically a high level design can be complex with infinite loops resulting indefinite many paths. However, in practice such designs are hard or impossible for HLS to generate RTL due to synthesizable limitations. An alternative way is to check the functional equivalence with a subset of paths with a high code coverage. For most of fixed loop or acyclic designs, full coverage is practical to achieve. In our flow we use KLEE to explore the behaviour C module to guarantee high code coverage given a reasonable time budget. KLEE can achieve high code coverage using finite program paths proved by various papers and studies, and the Verilog flow generates all the corresponding paths that can match all those C paths in the RTL counterpart. So the flow can terminate with a confident result.

4.4 Experimental Results

We have built our prototype to implement HLS functional verification flow based on LegUp 4.0 with LLVM 3.5, KLEE 1.3.0 with LLVM 3.4 ,and Yosys. The verification target is the Verilog code generated by LegUp for a C program. KLEE exploits logic expressions for return values of the C program, while Yosys encodes the circuit netlist to SMT-LIBv2 language for satisfiability testing. Several testcases have been analyzed to verify the validity of our framework on whether the logic flow generated by high-level synthesis tool, LegUp as an typical example here, is strictly compatible with the counterpart in C. All experiments were performed on a single core Intel i7-950 CPU machine with 8GB memory and 64 bit Ubuntu 16.04 operating system.

We start from fundamental inner product, FIR filter and FFT cases where data are fetched from RAM by memory controller in specific order. As fetching order is usually manipulated

by designers as an optimization strategy for higher throughput or less communication, it is imperative to verify the correctness of data traffic according to various demands. Meanwhile, such testcases contain a bunch of adders and multipliers, which are two basic components in numerical computation, so that they are also the simplest cases to effectively evaluate the scalability of our method when applied to numerical computation. We trace the memory address during the cycle-based symbolic execution of target circuit so as to bind the memory controller input/output to corresponding reference to external array in C program. Table 4.2 shows our flow has a fast run time and scales well with the rise of the number of total operations performed by the synthesized circuit. IR LOC is the number of LLVM-IR instructions symbolically executed by KLEE, which straightforwardly indicates the size of the testcase. Verilog lines would not be small with the basic functional modules describing the circuit, including dual-port ram and memory controller. We can also observe the Verilog lines are the same for designs with similar structure, for LegUp tends to reuse resources for conciseness as a default option. Therefore, the number of Verilog lines synthesized by tools is also a good indicator for the complexity of the RTL design. In FFT, the inverse flag is treated as a symbolic value. Thus two execution paths under different constraints, namely FFT result and its inverse one, can be captured by both RTL and C symbolic execution and successfully matched with each other.

We further show our flow running on convolutional neural networks (CNN), which are huge numerical models with convolutions. We first apply our method to convolutions with various input size and then evaluate the convolutions of different layers in Lenet-5 synthesized by LegUp tool to show the effectiveness of our method. We have fully verified the two convolution layers in this five-layer Lenet, and two more matrix convolution testcases, which is shown in Table 4.3. We observe that all testcases need more than 1500 clock cycles to finish circuit execution, and the first convolution layer needs 78840 clock cycles. All testcases finish execution with reasonable run time, and show good scalability.

We finally select floating point benchmark from CHStone benchmark [HTH08] to show the compatibility to more complicated cases. Those programs are double-precision floating point addition programs containing multiple execution paths. The signed flag and exponential part

Table 4.2: inner product and FIR results. Upper, middle and lower blocks show inner product, FIR and FFT results, respectively. SE means Verilog symbolic execution.

Array size	Finish clock	SE time(s)	Match time(s)	Verilog LOC	IR LOC
200	804	0.48	0.05	899	3814
500	2004	1.37	0.08	899	9514
1000	4004	3.54	1.83	899	19014
2000	8004	5.35	3.08	899	38014
200	1004	0.97	0.09	924	4214
500	2504	2.46	1.23	924	15014
1000	5004	3.32	1.58	924	21014
2000	10004	8.18	3.26	924	42014
5000	25004	25.86	24.63	924	105014
32	308	0.26	1.7	795	66109
64	664	0.45	13.57	795	157099

Table 4.3: Convolution results. At the third row and the fourth row we verify the convolution in Lenet-5 for the first and the second convolution layer.

Input size	Kernel size	Finish clock	SE time(s)	Match time(s)	Verilog LOC	IR LOC
16x16	3x3	1802	5.17	0.61	2054	50577
32x32	3x3	8170	18.84	1.41	2054	231677
32x32	5x5x6	78840	759.96	438.45	4150	2330466
10x10	5x5x10	6490	41.94	11.66	4214	2738090

is first extracted for two operands, and then all the bounded cases are taken care of by the benchmark program with an output concatenated by all valid segments. We collect the path constraints accordingly along the symbolic execution, where path constraint is incrementally recorded when a valid new execution state is forked for following execution. Then in the

matching stage, corresponding path constraint is asserted when the path is searching for its counterpart on the other side. We observe that Verilog code generated from HLS is much more complicated than C code at this case, this is due to the fact that HLS uses finite state machines to represent state goes through different branch conditions. As shown in Table 4.4, all C execution paths find its counterpart in Verilog. In *dfadd*, matching time is longer than symbolic execution time because the matching process is an enumeration process. An optimization method of avoiding this enumeration remains as the future work.

Table 4.4: Floating point benchmark results.

Benchmark	Finish clock	SE time(s)	Match time(s)	Verilog LOC	IR LOC	Path
dfadd	59	29.46	75.84	6068	2894	64
mips	83	2577.03	152.69	4195	1464	34
dfmul	129	165.15	0.69	10322	849	7
dfdiv	185	43.30	6.86	13359	1998	23
dfsin	437	753.25	15.42	28614	17855	17
sha	1488	139.37	83.5	7047	6867	1
motion	8198	150.53	2.80	34074	904	1

4.5 Conclusion And Future Work

In this paper, we propose a pure dynamic execution functional verification full flow for HLS. Specifically, our verification flow first runs light-weight symbolic execution on both C code and HLS generated Verilog code to generate SMT execution traces. Our on-the-fly lightweight symbolic execution on Verilog side is implemented on Verilog's SMT representation obtained by Yosys synthesis tool. Then we perform optimization on module input symbolization and valid datapath identification from HLS code generation perspective to greatly reduce the complexity for SMT solver. Considering HLS binds the same operation to an identical set of hardware, we perform one-on-one trace matching by abstracting identical operations on C and

Verilog traces. To the best of our knowledge, it is the first time that a pure dynamic approach is applied on HLS functional verification without transforming one side of the source code to the format of its counterpart before verification. Extensive experiments verify the validity and effectiveness of the flow. As for the future work, we will continue to optimize on the flow to reduce the runtime for the verification process to make it more promising and practical. We will also search for more efficient trace matching algorithms to avoid enumeration.

CHAPTER 5

Probabilistic Model Checking and Scheduling Implementation of Energy Router System in Energy Internet for Green Cities

5.1 Introduction

Energy crisis and carbon emission have become two seriously concerned issues in green cities [Che07,BZC15,ENS17] recently. As a feasible solution, Energy Internet (EI) [WYY17,ZYX16] has aroused global concern since it has been proposed. EI is a new power generation developing a green vision of evolution of smart grids into the Internet. Its organization is shown in Fig. 5.1. The key device to compromise EI is energy router (ER) [JWW16]. ER communicates with users similar to an Internet router, thus can perform immediate communication and control according to real-time user status to achieve green efficient energy management.This is vital to realizing green cities [OF15, MTG00].

A variety of researches have been done on the ER and green city topics. H. Zhang *et al.* [HWC17] proposes a multi-tier fog computing model with large-scale data analytics service for smart cities applications. W. Wang *et al.* [WLF17] proposes a system-level stability evaluation model in the Energy Internet based on a critical energy function to explore small disturbance stability region. W. Zhong *et al.* [ZYS16] presents a demand response model of vehicle-to-grid (V2G) mobile energy network in which the EVs generally move across different districts represented as network nodes. K. Wang *et al.* [WHL17] provides a survey to introduce EI communication for sustainability. Y. Zhang *et al.* [ZYX11] introduces architecture, standards, and QoS improvement for home machine-to-machine networks. K.

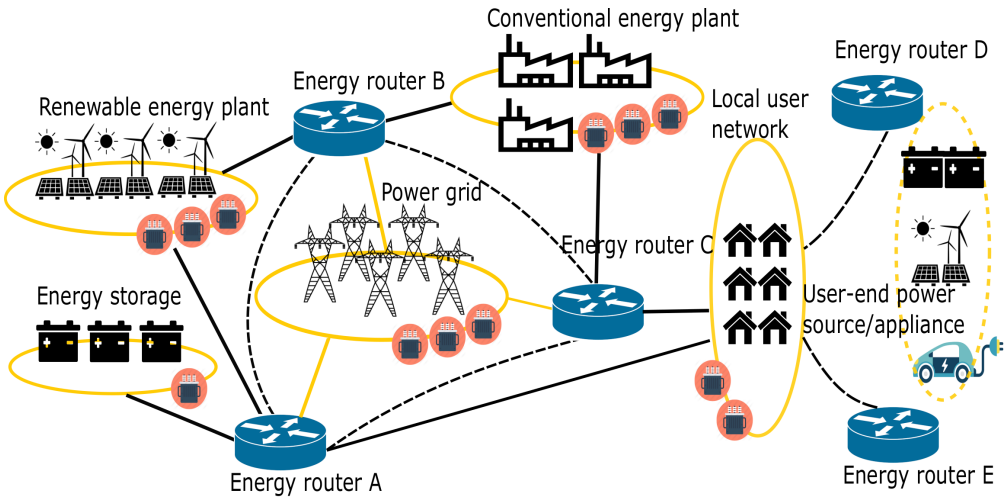


Figure 5.1: ER system in energy Internet for green cities

Wang *et al.* [WLF17] evaluates EI system's stability using big data analytics. Y. Zhang *et al.* [WWH17] proposes a big data computing architecture for smart grid analytics, which involves data resources, transmission, storage, and analysis. L. Xin *et al.* [XDS15] proposes design and applications of ER to realize EI. P. Yi. *et al.* [YZJ16] deploys ERs in an energy internet based on electric vehicles. It also provides feasibility analysis for applying ERs from power marketing, district heating, and control perspective. M. Behi *et al.* [BAJ11] designs a wireless green ER platform for controlling and scheduling of energies in energy efficient buildings. In the ER design and power transfer area, A. Sanchez-Squella *et al.* [SOG10] considers green efficient energy transfer among subsystems connected by the ERs. J. Miao *et al.* [MZK17] formulates the steady-state power flow model of the ER embedded system network and related optimal power flow formulation to optimize power system operation. S. Hambridge *et al.* [HHY15] introduces an economic based energy routing strategy utilizing energy storage to reduce consumption of grid power. However, all the work above lacks the consideration of soundness and completeness of the system design objective.

Formal verification proves the correctness of system with respect to a certain formal specification or property. For example in software field, many testing tools exploit formal verification in bug finding [GHM16]. One of the major approaches in formal verification is model checking [BCM90], which formulates system as transition graphs. It exhaustively traverses the graph, and verifies whether the model satisfies the formula representing the property. In real world, systems are inherently probabilistic, thus quantitative properties

are of the greatest interest to verify in addition to logic properties. Probabilistic model checking [KNP07] formulates systems into probabilistic transition models such as discrete-time Markov chains (DTMC), continuous-time Markov chains (CTMC), and Markov decision processes (MDP). A quantitative logic property is then applied to the model to check the result, and return a counter-example if a property is not satisfied.

In this paper, we propose a probabilistic model checking method to ER based system design, and monitor ER based system's running behaviour via our scheduling scheme. Specifically, we first propose a CTMC model on an ER based system containing multiple ERs to check the reliability of the system operation. Then we propose a CTMC model on a green ER based subsystem to perform model checking on its reliability and communication count properties. To apply all the communication functions into the real scenario for green cities, we propose an MDP electricity trading model, and model check quantitative properties on the service requester's cost and the service provider's loss. We also propose an energy scheduling simulation scheme for ER based system. In this scheme, we divide a load demand curve into multiple time windows, and then project each demand in a time window into a cloudlet in cloud computing, and then the energy scheduling process in ER based system is projected to host allocation process in cloud computing. We define our own host allocation policy to complete the scheme.

The contributions of this paper are summarized as follows.

- We introduce CTMC and MDP state machines to model ER based systems. To the best of out knowledge, it is the first time that formal verification technique is applied to ER based systems.

- We project the energy scheduling of ER based system into cloud computing area, and implement a tool to observe the performance of ER based systems due to the similarity of these two areas. It is the first time that a cloud computing tool is tailored to suit the need for ER based systems.

- We consider both electricity price and line loss during power transmission during the selection of power service providers. Extensive experiment verifies the effectiveness of

the proposed models and the monitoring scheme.

The rest of the paper is organized as follows. Section II introduces preliminary knowledge. Section III and IV introduce probabilistic model checking on architecture modeling and trading behaviour modeling of ER based system, respectively. Section V mentions scheduling for ER based system. Section VI shows experiments and results. Section VII concludes the paper.

5.2 Preliminary

5.2.1 Continuous-Time Markov Chain

Continuous-time markov chain (CTMC) is an automata in continuous time domain that preserves Markov property [KNP07].

Definition 1. *(Continuous-Time Markov Chain) A continuous-time markov chain C is a 4-tuple (S, s_{init}, R, L) where S represents a finite set of states, $s_{init} \in S$ is the initial state, $R : S \times S \to \mathbb{R}_{\geqslant 0}$ is the state transition rate matrix, and $L : S \to 2^{AP}$ is a labelling function which assigns to each state $s \in S$ a label with atomic propositions.*

The transition rate is used as the parameter of the exponential distribution. The probability triggered to make state transition before t time units for a rate r is $1 - e^{-rt}$.

As for model checking for the CTMC, Continuous Stochastic Logic (CSL) [KNP07] is widely used to specify properties.

Definition 2. *(Augmented Continuous Stochastic Logic) The state formula Φ and the path formula ϕ of Augmented CSL is defined by:*

- $\Phi ::= true|a|\neg\Phi|\Phi \wedge \Phi|P_{\sim p}[\phi]|S_{\sim p}[\Phi]$
  $|R_{\sim r}[I^{=t}]|R_{\sim r}[C^{\leqslant t}]|R_{\sim r}[F\Phi]|R_{\sim r}[S]$
- $\phi ::= X\Phi|\Phi \cup^{I} \Phi$

where a is an atomic proposition, P is the probability operator for all paths, S is the steady state operator, R stands for the reward function, I in Φ is a immediate time stamp, C stands for accumulated time, F stands for future states, X stands for the next state, I in ϕ is a interval of $\mathbb{R}_{\geqslant 0}$, $\cup$ is until operator, $\sim$ is a logical operator $\in \{<, \leqslant, >, \geqslant\}$, $p \in [0, 1]$, $r, t \in \mathbb{R}_{\geqslant 0}$.

Given a CTMC model C and a CSL formula ϕ, CTMC model checking outputs a set of states $Sat(\phi) = \{s \in S | s \models \phi\}$. The major task is to calculate probabilities or rewards for $P_{\sim p}[\cdot]$, $S_{\sim p}[\cdot]$ and $R_{\sim r}[\cdot]$, which is discussed in [KNP07].

5.2.2 Markov Decision Process

Markov Decision Process(MDP) is a non-deterministic state transition automata that preserves Markov property [KNP07].

Definition 3. *(Markov Decision Process) A Markov Decision Process M is a 4-tuple $(S, s_{init}, Steps, L)$ where S represents a finite set of states, $s_{init} \in S$ is the initial state, $Steps : S \to 2^{Act \times Dist(S)}$ stands for the state transition probability function where Act is a set of actions and $Dist(S)$ is the set of probability distributions over the set S. $L : S \to 2^{AP}$ stands for a labelling with atomic propositions.*

Probabilistic computation tree logic (PCTL) and its augmented version serve as the backbone in model checking an MDP.

Definition 4. *(Augmented Probabilistic Computation Tree Logic) The syntax of Augmented PCTL is shown as follows:*

- $\Phi ::= true | a | \neg\Phi | \Phi \wedge \Phi | P_{\sim p}[\phi]$
  $| R_{\sim r}[I^{=k}] | R_{\sim r}[C^{\leqslant k}] | R_{\sim r}[F\Phi]$
- $\phi ::= X\Phi | \Phi \cup^{\leqslant k} \Phi | \Phi \cup \Phi$

where a is an atomic proposition, P is the probability operator for all paths, R stands for the reward function, I in Φ is a immediate time stamp, C stands for accumulated time, F stands

for future reachiable states, X stands for the next state, $\cup$ is until operator, $\sim$ is a logical operator $\in \{<, \leqslant, >, \geqslant\}$, $p \in [0, 1]$, $r \in \mathbb{R}_{\geqslant 0}, k \in \mathbb{N} \cup \{\infty\}$ is a until bound.

The main unique task for augmented PCTL is to evaluate the probabilistic operator P and the reward operator R. Evaluating $P_{\sim p}[\phi]$ is reduced to evaluate either minimum probability of ϕ holding or maximum probability of ϕ holding depending on the $\sim$ operator over all the paths. If $\sim \in \{<, \leqslant\}$ then the maximum probability is calculated to check ϕ holding for $s \models P_{\sim p}[\phi]$, otherwise the minimum probability is calculated. The calculation of reward follows the similar idea.

5.2.3 Energy Router Based Subsystem Architecture

The architecture of an ER based subsystem is shown in Fig. 5.2. As stated in [XZW11], the router mainly contains a communication module and a control module. Ports provide plug-and-play interfaces controlled by the router to convectional power source, renewable power source such us wind turbine and solar cells, as well as energy storage and load. Router communicates to each port through wired cord or wireless channels. Power electronics devices such as solid-state transformers, converters and inverters electrically connect all the ports. In our work, the main modeling focus is on the communication and control module instead of the power electronic devices. The functionality of the ER falls into user-level and gate level. At user-level, main functions include user attachment/detachment, service request/termination and status update. At gate-level, the ER performs communication between not only its local users but also other ERs. Real time energy control, management and trading are thus available through these basic functions.

5.3 Architecture Modeling of The Energy Router System

We propose two CTMC models to describe the green energy router system architecture. The first model describes the behaviour of a system consists of multiple ERs shown in Fig. 5.1, and the other is a model on the operation of an ER based subsystem shown in Fig. 5.2. In a poisson

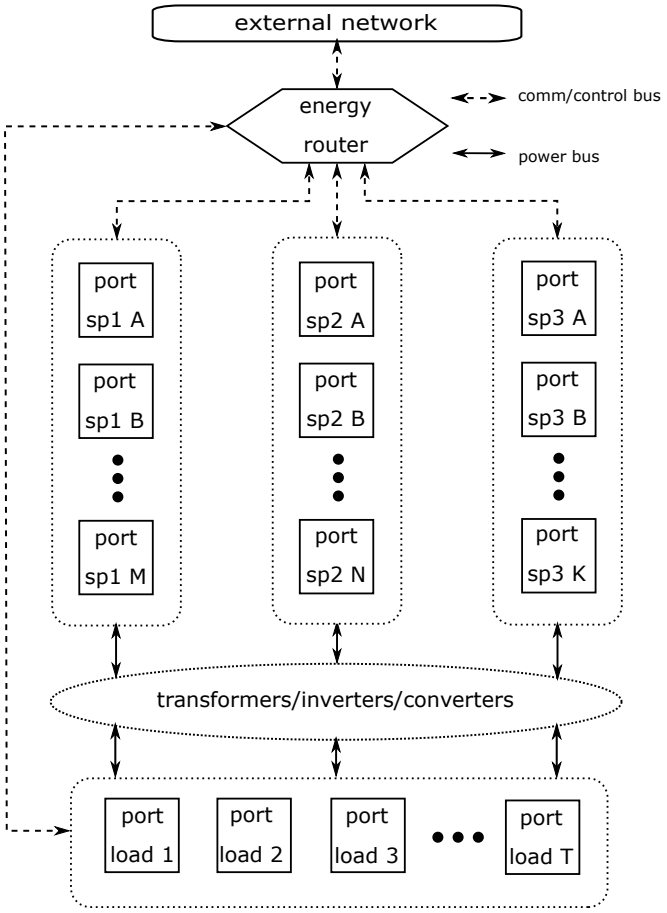


Figure 5.2: Architecture of single ER-based subsystem

process, the length of the inter-arrival time follows exponential distribution. Exponential distribution also describes naturally the time for a continuous process to change time. We choose CTMC because we assume state transitions follow exponential distribution, and the state transition is memoryless, and related work on dynamic energy management [QP99] models the behaviour of a power-managed system as a CTMC model.

5.3.1 Modeling of The Multiple Energy Router System

A multiple-ER system contains multiple ERs and multiple associated ports.We generalize Fig. 5.1 to a controller module, an ER set module representing a set of ERs, and a port module representing the set of ports controlled by the routers. Modeling every operation in each components results in the scale of the model growing exponentially, thus causing the state space explosion. Instead of directly modeling each component's broken and recovered

state using 2^n states, we set the number of broken components c_{down} as the state variable because we do not care exactly which component is broken from the reliability perspective. c_{down} increases and decreases according to the failure and recovery rate. We assume the failure rate is dependent on the number of failed components. This is because once a failure happens, all the similar components will be checked to prevent the same issue from happening again easily. Thus we set an coefficient to regulate the failure rate. The regulated rate r_{fail} is given by

$$r_{fail} = e^{\frac{-comp_{down}}{comp_{total}}} \times MTTF, \tag{5.1}$$

where $comp_{total}$ is the total number of components, and $MTTF$ is the mean time to failure of the component. This abstraction will reduce the state space to the order of n^2.

The controller module serves as the hub of the system, it contains three operation states– **SLEEP**, **BUSY** and **BROKEN**. In normal operation mode, the controller switches between **SLEEP** and **BUSY** if the number of down routers and ports are less than the maximum threshold. Controller itself transits to a **BROKEN** and reverts to **SLEEP** after repairing according to its rate.

The state transition of the three modules are synchronized through actions, i.e., one action triggers state transitions in multiple modules simultaneously. The failure of the system is defined by any of the component that is unable to meet the normal operation requirement. We label the system failure by “down" as

$$down = s_{ctr} == 0 \vee f_p > 0 \vee f_r > 0, \tag{5.2}$$

where s_{ctr} is the controller state, f_p and f_r are the number of failed ports and routers, respectively. The reliability property $S =?[\text{“}down\text{”}]$ is thus the main property we are interested in.

5.3.2 Modeling of Single Energy Router Based Subsystem

We consider an ER based subsystem shown in Fig. 5.2 whose control and communication functionality is achieved through wireless communication. Thus in our model a local cell

Table 5.1: Local state variable for single ER-based subsystem

Variable Name	Value	Explanation
s_p	0	broken
(port	1	idle
state)	2	sending message and waiting response
	3	receiving message and processing
s_r	0	broken
(router	1	idle
state)	2	sleeping
	3	processing incoming message
	$\geqslant 4$	send message back to ports
q_{size}	0-maxsize	number of requests in the queue
in_x	0-1	whether X is in the cell

module with a guarded channel [HMP01] to reduce the handoff dropping probability is introduced as the communication medium. A handoff drops when a device enters a new cell but the new cell has no channel to provide. Guarded channels are reserved exclusively for the handoff calls. In this module the number of calls in the cell *nLocal* and existence of the ports and routers in_x are set as state variables. We set the router as the handoff call and all the other ports as regular calls. Enter or departure of these calls will increase or decease the *nLocal* and change the in_x correspondingly.

In port module we define four operating states listed in Table 5.1. The normal duty cycle circulates among port **IDLE** ($s_p = 1$), **SEND** ($s_p = 2$) and **RECEIVE** (s_p=3) according to the transition rate. One of the guards for each state transition is $in_x = true$. The port may run into **BROKEN** from any states. A repair action resets **BROKEN** back to **IDLE**.

The router module receives all the messages and requests from the ports, processes all those messages and requests, and then sends results back to all the ports. To reduce computing complexity of the model checker, we choose to lump the service model as a sequential queue.

Thus we embed an M/M/1 service queue module to model the service queue inside of the router. Different ports send messages to the queue according to its own rate while increasing q_{size} by 1. The router first transits from **SLEEP** ($s_r = 2$) to **IDLE** ($s_r = 1$). After checking the local cell module for a true value of in_{router}, it fetches one request from the queue, and then transits to one of the sending back states. This is a non-deterministic transition whose transition probability is in proportional to the request rate. We then have the following lemma.

Lemma 5.3.1. *The state transition probability of a set non-deterministic of n choices in the router is given by* $req_i/\Sigma req_i$.

Proof. The controller sends message back to head of the queue, which is the index of the ports sending out the request with the minimum time. Since all the ports follow a exponential distribution of parameter req_i, the minimum time distribution parameter is given by Σreq_i. For a single port then the probability is $req_i/\Sigma req_i$. □

We then set the transition rate in proportional to this probability to reflect the right transition distribution.

The q_{size} is decreased by 1 after the choice is made, and s_r goes back to idle. **IDLE** transits to **SLEEP** according to a preset rate. The router may run into **BROKEN** state from any of the states mentioned above. A repair action then resets the state into the sleeping state.

The main properties to be checked are listed in Table 5.2. One crucial property of for this system is the reliability. The failure of system is defined by any of the component that is under broken state or disconnected from local cells. We thus label the system failure as "down" by

$$down = s_r == 0 \vee s_{ps} == 0 \vee in_x == 0, \tag{5.3}$$

and intend to observe the probability of failure in the long run. Another property we are interested in is energy consumption during the system operation. We generalize it to

Table 5.2: Single ER-based subsystem properties

Property	Explanation
$S =?[\text{“}down\text{”}]$	The system is down in the long run
$R\{\text{“}comm\text{”}\} =?[C \leqslant T]$	The total number of services that the router has performed until time T

the measurement of communication operation count by setting a reward of 1 to all the communication actions and labeling the those actions with *comm*.

5.4 Modeling of Energy Router Subsystem Based Electricity Trading

With the real time communication support of ER, green energy management and trading can be applied. Based on the ER subsystem, we propose and model check an MDP electricity trading scheme which extends the general trust model in [AB12] by considering power demand and demand response as a real world application of the ERs. We choose MDP because state transitions are made based on decisions.

The whole scheme models the trading operations for n hours. Every 1.5 hrs the requester *req* can choose whether to generate a load demand or not. Once a provider accept the request, *req* has the non-deterministic choice on whether to accept and whether to actually pay for the service with a reference of its own price threshold. Prices should be paid after the service begins but before next request decision. Within n hrs *req* has to successfully receive and consume power k times.

The proposed model contains one service requester *req* and multiple service providers offering electricity from green energy sources. We assume that collaboration between requesters will not gain more reward since in between a group of load ports there are no additional devices to perform power redistribution. We also assume any supplier has enough power to provide once it agrees to provide service, and line loss for all the suppliers are the same to keep the scale the proposed model reasonable.

Table 5.3: Local state variable for a service requester

Variable Name	Value	Explanation
s_{req}	0	idle
(requester	1	a load request is generated
state)	x1	requester is requesting sp_x
	x2	sp_x accepts the service and issues a price
	x3	sp_x refuses the service
$sp_{x,r}$	0-1	whether sp_x refuses service at this time stamp

5.4.1 Requster Modeling

Table 5.3 lists all the local state variables for a requester req. When a load request is generated ($s_{req} = 1$), req chooses one service provider randomly, setting $s_{req} = x1$. If a service provider sp_x agrees to provide the service after checking req's trust level T_x, a synchronized action takes place on both the provider x and the req along with a price P_x calculated, and thus $s_{req} = x2$. Otherwise another synchronized action sets $s_{req} = x3$. The reputation of the requester req at sp_x is denoted by tr_x, representing the extend of provider x trusting req. The trust level T_x is a linear combination of tr_x at sp_x and other service providers, which is given by

$$T_x = \alpha \cdot tr_x + (1 - \alpha) \cdot rec_x, \tag{5.4}$$

where

$$rec_x = \begin{cases} tr_x, & \text{if } know_y = 0 \ \forall y \in Y \\ \frac{\Sigma ite(know_y, trust_y, 0)}{\Sigma ite(know_y, 1, 0)}, & \text{otherwise} \end{cases} \tag{5.5}$$

is the recommendation factor from other service providers, and $know_y$ is true if there exists the transaction history on sp_y. ite is a “if-then-else" expression.

A uniqueness in power system is that power demand fluctuates over time. Thus we set P_x at different timestamps in accordance with the fluctuation. One more factor needed to be considered is the req's reputation, and we set an inverse scaling factor on the time-specific price to reflect the reputation. Thus the price for a requester req once a provider x agreed to

serve is given by

$$P_x = C \cdot (\frac{\beta}{tr_x} + 1 - \frac{\beta}{D}) \cdot Pr_{demand}(t), \tag{5.6}$$

where $C, \beta, D \in \mathbb{R}_{\geqslant 0}$, and $Pr_{demand}(t)$ is a piecewise constant demand approximation function whose data is given in [HS11]. C, β, D constitutes the conversion rate from demand and price. β, D takes trust into account in an inverse proportional fashion, and C is a conversion factor with a unit of $\$/kwh$.

Once P_x is issued, the req compares P_x with its own target price P_{target}. P_{target} is in proportion to the power demand over time regulated by a parameter γ, which is global to both requester and providers. req has two non-deterministic choices when a low P_X is issued. One is to get the service and pay, or get the service without payment. If req finds P_x is higher than expected, it can withdraw the load request in addition to the previous two choices, which reflects the demand response.

If the service provider refuses the service request, then the corresponding $sp_{x,r}$ flag is set to 1. The req continues to query other unvisited service providers until all the providers refuse to serve. $sp_{x,r}$s are reset to 0 before the next time stamp begins.

5.4.2 Service Provider Modeling

Table 5.4 lists all the local state variables for a service provider sp_x. When a synchronized service request action arrives, sp_x compares the trust level T_x with th_x. An accept action sets s_{sp} to 1, $know_x$ to 1, and changes s_{req} in req when $T_x \geqslant th_x$, or a reject action sets s_{sp} to 0 otherwise. It is reset to 0 after the request get processed before the next timestamp.

Getting paid service gains reputation bonus on req, and reputation penalty if the service is not paid. Accordingly, the trust threshold of the req varies depending on the choice made. Unpaid service results in a higher threshold, and paid service results in a reasonable threshold. We set small penalties and threshold increase on lower P_xs, and high penalties and threshold increase on higher P_xs. All the actions are synchronized with req, thus when corresponding conditions for an action on req and sp_x hold, both states changes simultaneously.

In order to guarantee the load having k times power to consume, we set a have-to-pay

Table 5.4: Local state variable for a service provider

Variable Name	Value	Explanation
s_{sp} (service provider state)	0	idle
	1	processing request
$know_X$	0	sp_x has no transaction record for req
	1	sp_x has transaction record for req
th_x	1-10	threshold for sp_x to refuse service
tr_x	1-10	reputation of req to spX

Table 5.5: Green electricity trading properties

Property		Explanation
$R\{"cost"\}min$ $?[F(time \geq max\_time)]$	=	req's minimum cost to get required number of service
$R\{"loss"\}max$ $?[F(time \geq max\_time)]$	=	sum of spX's maximum loss to provide required number of service

mode for req and a have-to-serve mode for the provider when the remaining time is limited. req stops request power when it is been served for k times.

Table 5.5 lists the quantitative properties we check for the MDP model. From the requester perspective, req wants to get serviced with the minimum amount of money. In our model, we label all the paid actions as "cost", and accumulate the cost until the end. From the service provider perspective, providers intends to estimate their maximum loss before participating the service. In our model, we label all the unpaid actions as "loss", and accumulate the loss until the end.

5.5 Energy Router System Scheduling

Probabilistic model checking for ER based system shows a feasible way of calculating safety probabilities and optimal costs. To observe single behaviour or average performance of ER based system for better decision-making purposes, a scheduling platform is needed. This goal is similar to cloud computing since both cloud computing and ER based system perform task scheduling and allocation. In this section, we implement a scheduling simulation tool on top of the cloud computing tool *CloudSim* [CRB11] to utilize the similarities.

A typical *CloudSim* system compromises of a data center, a data broker, multiple cloudlets, virtual machines, and hosts. The *data center* is an abstract container that manages all the physical computing sources. Those computing sources are called *hosts*. The data center *broker* is responsible for sending multiple computing tasks to the data center, and acknowledge its completion. Each computing task is called a *cloudlet*, and usually those cloudlets can be bound with a computing resources designated by the cloud service provider. We call those computer resources a *virtual machine*. The broker sends all required virtual machines into the data center as well. The data center allocates virtual machines into different hosts according to user defined allocation policies.

Since both cloud computing and ER based system perform task scheduling and allocation for some specific entities, an interesting question is that can ER based system projected into cloud computing based platform? In this section we provide a positive answer. We first abstract the energy router along with all the power sources as a data center. Then, we divide a load demand $demand_i$ into multiple piece-wise constant time windows. Each time window of $demand_i$ is regarded as a set of cloudlets that starts computing at different specific time stamps. A virtual machine is directly bound to a cloudlet to guarantee the the cloudlet can finish computing within the scheduling interval. We label the virtual machine as vm_i^j representing for $demand_i$ at a time window order j. Thus $demand_i = \{vm_i^1, vm_i^2, ..., vm_i^N\}$. A host *host* then becomes a power source provider naturally. Then question for the ER based system becomes for each $demand_i$ at the beginning of each time window j, vm_i^j should be allocated to which $host_k$ via the energy router according to designated policy.

5.5.1 Implementation of ER based System Simulation

From the problem formulation, we need *CloudSim* to have the following three properties.

- **All virtual machines should be able to start at user specified time stamp.**
- **A virtual machine should be freed from its host and destroyed once the task is completed.**
- **A new host allocation policy is required to make load choose a feasible power service provider.**

5.5.1.1 Modifying Virtual Machine Starting Time

We modified the interface of *CloudSim* to suit all the needs for ER based system simulation. In *CloudSim*, all the cloudlets stored in an array are submitted by the broker at the beginning of the experiment via the function call *sendNow*. We created a new class *ERCloudlet* specifying the start time the cloudlet to be schduled, and replaced *sendNow* with another API *send* in the original source code that can specify the starting time of a cloudlet to be executed. Instead of submitting cloudlets by array at time 0, the broker in our implementation submits cloudlets one by one at their designated starting time.

5.5.1.2 Modifying Virtual Machine Destroy Time

The destroy process of all virtual machines in *CloudSim* happens when all the cloudlets completed their computing task. In ER based system, a cloudlet stands for the energy needed in a time window. Thus it should be finished before the time window ends. Consequently, the virtual machine should be deallocated from the host before next time window come. In our implementation we destroy virtual machines one by one, and shifted the destroy process to an earlier stage when the broker acknowledges that the virtual machine finishes computing.

5.5.1.3 Host Allocation Policy for ER based system

Host allocation policy in the cloud computing defines which virtual machines is allocated to which host. Under the scenario of ER based system, this policy assigns vm_i^j representing for load demand to $host_k$ that stands for a power service provider. To be more specific, a load requests power to all power providers through the energy router during the start of each time window, and then each power provider sends back a price. We set the pricing scheme to be time dependent, and also trust dependent if the trust system is enabled. The load chooses which power station to provide power supply according to user defined criteria, and also chooses to pay or not to pay after receiving the desired power if the trust system is enabled. The load's trust increases if it pays for the demand generated, otherwise the trust decreases.

5.5.1.4 Trust based Power Transaction

The scheme for building trust system is totally customized. Our implementation of trust system is almost identical to section IV. The trust query scheme whose trust evaluation is identical to Eq. (5.4) and Eq.(5.5). The pricing scheme for service provider is identical to Eq.(5.6). The only difference is that we disabled spx, r but setting a load's trust to minimum instead. This guarantees every vm_i^j will receive power service.

A load will choose a power provider not only considering the electricity price, but also line loss. Line loss cost $loss_L$ defines the money wasted on the energy loss transmitted from a power plant to the load. Power service provider with longer distance incurs higher $loss_L$. For simplifying computation complexity, we define the total payment P_{total} to get required amount of energy to be approximately $P_x + loss_L$. The load checks the cost from all the hosts, and then chooses the service provider that has enough power with the lowest cost. Each node has its estimated price P_{est}, and we relate P_{total} and P_{est} via a coefficient f_{exp}. If $P_{total} > f_{exp} * P_{est}$, the load will have higher probability not to pay the service, otherwise the load will have lower probability not paying.

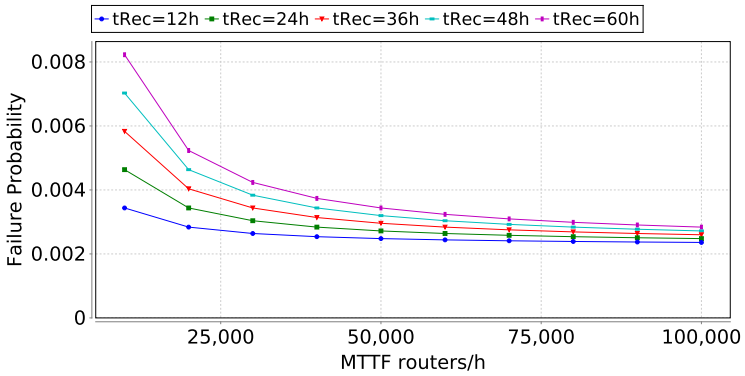


Figure 5.3: Multiple-ER system failure probability

5.6 Experiments and Results

We implemented all three models in the probabilistic model checking tool *PRISM* [KNP11]. Prism is a probabilistic model checker supporting modeling of DTMCs, CTMCs, MDPs, and PTAs. We also implemented our ER based simulation framework on top of *CloudSim*. All the evaluations and simulations are run on a desktop equipped with an Intel E5-2643@3.30G CPU and 128GB memory. The PRISM's cudd memory is set to be 4GB, and java memory is set to be 16GB.

5.6.1 Architecture Model Properties

The multiple-ER model contains 1 hub, 10 ERs and 1000 ports. It has 0.8k states and 2k state transitions. Configurable parameters include mean time to failure (MTTF) and recovery time for the controller, the ERs and the ports. Our main interest for this model is its reliability. We varied the MTTF and the recovery time for the ERs to verify the reliability property whose result is shown in Fig. 5.3.

We observe that the system failure probability goes down when the ERs have larger MTTF and faster recovery time, and an ER MTTF larger than $100,000hrs$ provides limited contribution to the system reliability because of the probability convergence. Curves for different MTTF settings for each ER will be bouded in this set of curves.

The ER based subsystem model contains one ER module, one local cell module, one service queue module, and three port modules. The number of channels in the local cells

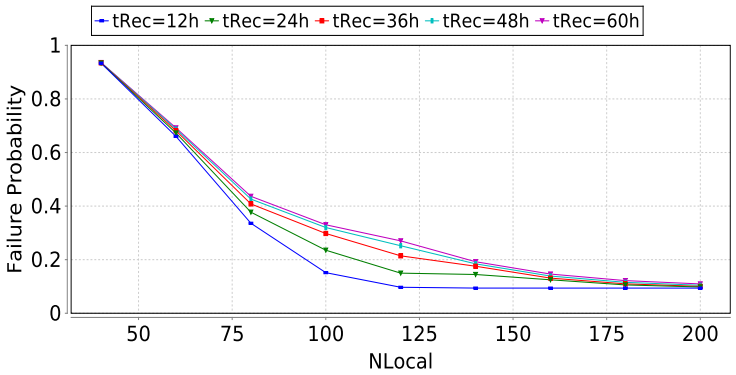


Figure 5.4: Single-ER based subsystem failure probability

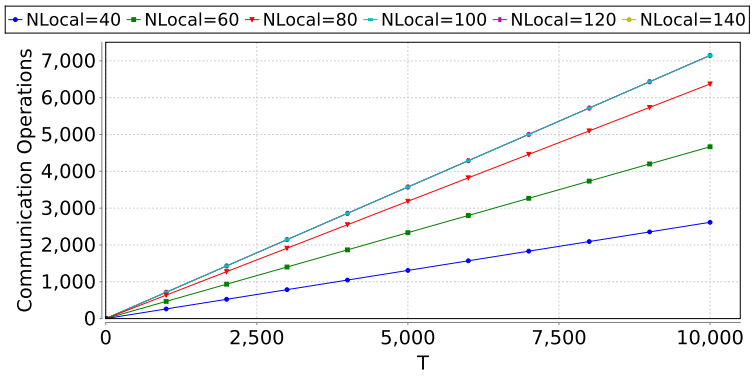
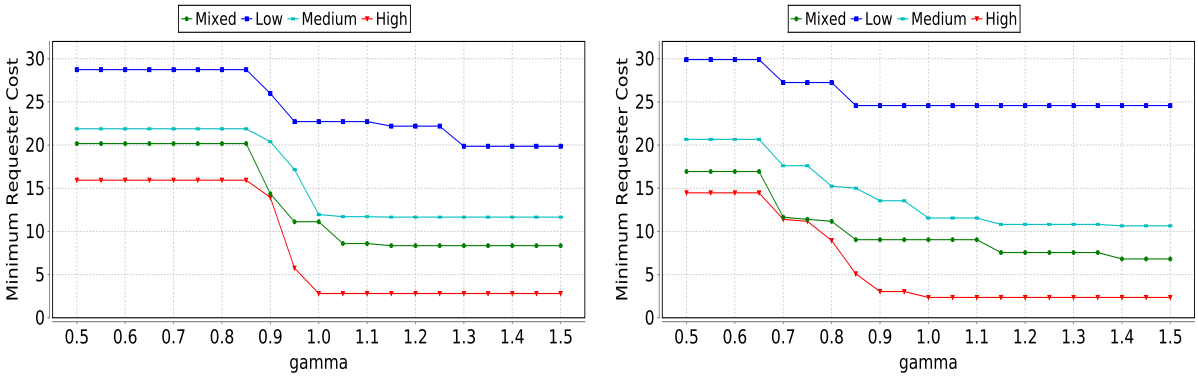
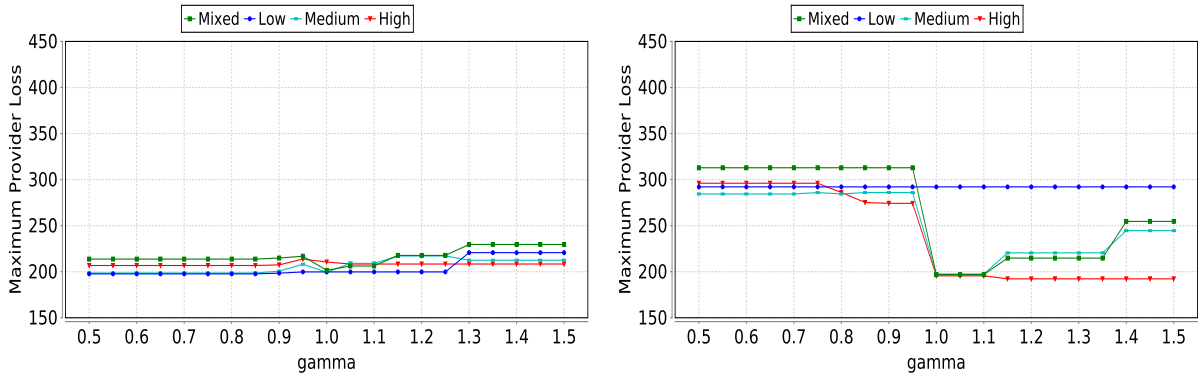


Figure 5.5: Single-ER based subsystem communication count



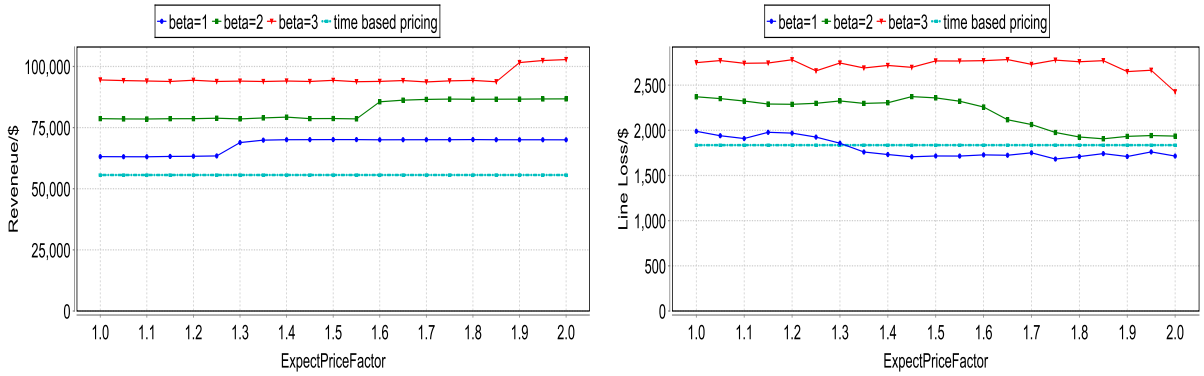
(a) $\beta = 1$ (b) $\beta = 3$

Figure 5.6: Minimum cost for the requester to get the required amount of services



(a) $\beta = 1$ (b) $\beta = 3$

Figure 5.7: Maximum loss for the provider to provide the required amount of services



(a) Average total revenue (b) Average total line loss

Figure 5.8: Average total revenue and average total line loss for ER based system

$NLocal$ and ER's recovery time are set as experiment parameters, where $NLocal$ varies from 40 to 200, and $tRec_{Router}$ varies from 12 hrs to 60 hrs. Average life time for the ER is 50000hrs. Average life time for all three ports are 1 month, 1.5 months and 2 months. Request rate for all three ports which are the inverse of the corresponding average request frequency are 0.015, 0.02 and 0.025. In the service queue module, the maximum capacity for the service queue is 50. In the local cell module, the arriving rate of new calls and hand-off calls are 49 and 21, and departure base rate is 1. In total the whole model contains approximately $14million - 54millon$ states and $50millin - 489millon$ state transitions depending on the number of local channels.

The model checking result for the reliability property is shown in Fig. 5.4 where $NLocal$ stands for the number of channels in the local cells. We observe the reliability of the local cell contributes the major part of the failure. Higher cell channel capacity results in lower failure probability. Faster recovery time results in lower failure probability as well.

We also checked the total router related communication operations property shown in Fig. 5.5 where T varies from 0 to 10000 seconds to get a taste of energy consumption under the current parameter setting. We observe a linear increment of number of operations over time, and a increment with more local channels at the same time. The operation number saturates when $nLocal$ is large enough.

5.6.2 Electricity Trading Model Properties

In this model, we checked properties from both the service requester and the service providers perspective. This model contains one service requester and three service provider. The requester is required to consume energy for 15 times during 48 time slots (72 hours). The trust level varies from 1 to 10. We set α to 0.5, $\beta \in \{1, 3, 5\}$, and γ varying from 0.5 to 1.5 with a step of 0.1. The full model contains $113million$ states and $194million$ transitions. We check the minimum cost from the service requester side, and the maximum loss property from the service provider side.

The minimum loss for the requester to get required services is shown in Fig. 5.6, where

the mixed (2/5/8), low (2/2/2), medium (5/5/5), high (8/8/8) stands for the requester's initial trust levels for all three providers. The run time for the experiment is 14 hrs. We observe a higher γ results in smaller cost, and the cost starts to drop earlier when β increases. It is also clear that the requester's strategy to get lower cost is to increase its target price, and the cost starts to saturate when the β is large.

The maximum loss for the all service providers is shown in Fig. 5.7, where the initial trust settings are identical to the minimum loss checking. We observe different behaviors on varying β and γ. When β is small, all four trust settings show similar increasing trends which is insensitive to the γ. The interpretation behind is that a small γ does not regulate the price tight enough, thus under all situations the requester will take similar strategy to pay as less money as possible to get serviced. The amount increases in high γ due to the decrement in the requester's cost. When β is large, low initial trust indicates the requester has only limited chances to get serviced. Thus it follows the same strategy regardless of the γ. Under other initial trust levels, a small γ results that the issued price is always higher than the target price. To guarantee the load consumption with the minimum money, the user will perform unpaid actions as many as possible. Thus the service providers suffers constant high loss. When γ is high, the issued price can be lower than the target price. This increases the choices for the requester to pay less, and the maximum loss gradually increases. However, all high initial trusts makes the optimal choices fixed again because every transaction will be consented by the providers. Thus the solution space becomes the same regardless of the γ.

5.6.3 Scheduling Behaviour of ER Based System

We built a ER based system containing three loads and three power service providers controlled by an energy router. The system contains three categories of input data, distance from each load to each power provider, a 5-day demand with a unit of MWh for each load with a time window of 1 hour length, a 5-day time dependent electricity price having a unit of $\$/MWh$ from a power provider with a time window of 1 hour length. The distance is generated by a random number generator, and the other two data comes from running data of ISO-NE [ISO].

We set a 0.15 probability that a load will refuse to pay the service if the price is higher than expected, and 0.05 if lower than its expectations. Then we varied β from 1 to 3 in Eq.(5.6) and $ExpectPriceFactor$ from 1 to 2 to observe the service providers total revenue and total line loss in dollars within 5 days. Since the load actions are stochastic, for each parameter setting we performed experiment 500 times, and returned their average outputs. We also measured the total revenue and total line loss using time based pricing only with no trust system applied as a base scenario to compare.

Fig. 5.8a shows the total revenue of all service providers. Comparing with the time based pricing, we observe that introducing the trust system in general increase the total revenue since a higher price will be applied to the load with low trust score. A higher *beta* value yields higher revenue since high β increases the price for low trust score load. We also observe that load's $ExpectPriceFactor$ has effect on the total revenue. The total revenue increases when $ExpectPriceFactor$ is around $1.3, 1.6, 1.9$ when β equals to $1, 2, 3$ respectively. This indicates that below those factors the prices issued from the service providers are in general satisfy loads' expectations, and thus less loads refuse to pay the price. When the factor is high, more loads refuse to pay for the power they got from the power provider, and thus bad trusts result in higher revenue for the providers.

Total line loss of the experiment is shown in Fig. 5.8b. We observe that line loss increases while β is increasing. This is because the higher price with higher β yields higher loss in terms of money even with the same loss in terms of energy. Higher $ExpectPriceFactor$ help reduce the line loss in general. Comparing to the no trust time based pricing case, $\beta = 1$ yields even lower line line when the $ExpectPriceFactor$ is high.

5.7 Conclusion and Future Work

In this paper, we applied probabilistic model checking to verify the energy router system design in Energy Internet for green cities. We proposed two CTMC models to describe operation of a multiple ER system and a single ER subsystem respectively, and presented their reliabilities and expected communication operations through probabilistic model checking. To apply

ER system functions into real scenario for green cities, we selected electricity trading as an example, and proposed an MDP model to describe the trading behaviour, and quantitatively model checked the minimum cost of the service requester and the maximum loss of the service providers. We also introduced an ER based system scheduling monitoring tool to observe the system's scheduling behaviour. We projected the power demand and supply transaction process to computing cloudlets in cloud computing, and then modified *CloudSim* to suit the need of ER based system transaction. As the future work, we will perform probabilistic model checking on more real functions in the energy router system, apply other formal verification techniques to verify the system design, and improve the monitoring tool to model more complex ER based system design for the future development of green cities.

CHAPTER 6

Summary

In this chapter, we conclude the dissertation by summarizing our contributions and presenting the future work.

In the software testing field, since concolic testing suffers from path explosion, we introduce the concolic+BMC algorithm that applies BMC locally targeting at loop-free code fragment during concolic testing to alleviate path explosion, and thus improve branch coverage. We have compared LLSPLAT with KLEE, using 10 programs from the Windows NT Drivers Simplified [SVC] and 88 programs from the GNU Coreutils used in [CDE08]. With 3600 second testing time for each program, LLSPLAT provides on average 13% relative branch coverage improvment on the programs in the Windows NT drivers simplified set, and on average 16% relative branch coverage improvment on 80 out of 88 programs in the GNU Coreutils set. The experiments show that the concolic+BMC algorithm increases branch coverage of the two test benchmark sets. we also adopt symbolic execution and automatic test case generation into embedded platforms. We use a client/server model to separate constraint solving and symbolic execution to greatly reduce the computation load in the embedded platform. Multi-threading in both client and server modules increases the efficiency of the Codecomb. Experiments on PC and Pandaboard embedded platform show the Codecomb can run on the embedded platform, and is able to find software deficiencies automatically.

We believe some future work can be achieved on top of our algorithm. Specifically, the implementation of LLSPLAT performs BMC encoding if there exists an acyclic graph containing a few merging basic blocks without function calls inside of the graph. We avoid encoding function calls because it may incur exponential blowup in the BMC formula generation. We would like to come up with a clever evaluation procedure that identifies “cheap" function

calls that can be encoded. In addition, solving functions of large scale may yield long time being spent in the SMT solver. We would also like to investigate whether there exists a low cost governed region overhead estimation method to make the selection of BMC regions more intelligent. As for Codecomb, we will continue improve Codecomb to make it suitable for more complex source code, and add support to more communication ports to make Codecomb suits more sorts of embedded platforms.

In the VLSI HLS filed, we propose a dynamic execution functional verification flow for HLS. Specifically, our verification flow first runs light-weight symbolic execution on both C code and HLS generated Verilog code to generate SMT execution traces. Our on-the-fly lightweight symbolic execution on Verilog side is implemented on Verilog's SMT representation obtained by Yosys synthesis tool. Then we perform optimization on module input symbolization and valid datapath identification from HLS code generation perspective to greatly reduce the complexity for SMT solver. Considering HLS binds the same operation to an identical set of hardware, we perform one-on-one trace matching by abstracting identical operations on C and Verilog traces. To the best of our knowledge, it is the first time that a pure dynamic approach is applied on HLS functional verification without transforming one side of the source code to the format of its counterpart before verification. Extensive experiments verify the validity and effectiveness of the flow. As for the future work, we will continue to optimize on the flow to reduce the runtime for the verification process to make it more promising and practical. We will also search for more efficient trace matching algorithms to avoid enumeration.

In the EI field, we applied probabilistic model checking to verify the energy router system design in Energy Internet for green cities. We proposed two CTMC models to describe operation of a multiple ER system and a single ER subsystem respectively, and presented their reliabilities and expected communication operations through probabilistic model checking. To apply ER system functions into real scenario for green cities, we selected electricity trading as an example, and proposed an MDP model to describe the trading behaviour, and quantitatively model checked the minimum cost of the service requester and the maximum loss of the service providers. We also introduced an ER based system scheduling monitoring tool to observe the system's scheduling behaviour. We projected the power demand and

supply transaction process to computing cloudlets in cloud computing, and then modified *CloudSim* to suit the need of ER based system transaction. As the future work, we will perform probabilistic model checking on more real functions in the energy router system, apply other formal verification techniques to verify the system design, and improve the monitoring tool to model more complex ER based system design for the future development of green cities.

.1 Preliminaries

Given a control flow graph (CFG) of a function, $BB_0, BB_1, \ldots, BB_n$ is a *path* of CFG if for each $0 \leq i \leq n-1$, (BB_i, BB_{i+1}) is an edge of CFG. Given an edge (a, b) of the CFG, we call a the *source* of the edge and b the *target* of the edge. A state s of a program is a function that maps each variable x in the program to a value in the domain of x. Given two states s and s', we denote $s \xrightarrow{BB} s'$ to be an execution such that by executing the instructions of BB with the initial state s, the execution ends up with the state s'. Occasionally, if we are not interested in s or s', we omit them and write $\xrightarrow{BB} s'$ or $s \xrightarrow{BB}$.

Given a formula ψ, an *assignment* m of ψ is a function that maps each variable x in ψ to a value in the domain of x. An assignment m is a *model* of ψ, denoted by $m \models \psi$, if ψ evaluates to *true* by m.

Given a set S of variables, a version map $\mathcal{V}$ is a *renaming* function that maps each variable $x \in S$ to a variable x_α for some $\alpha \in \mathbb{N}$. We write $\mathcal{V}(S)$ to be the set of variables $\{y \mid \exists x \in S.\ y = \mathcal{V}(x)\}$. Given a version map $\mathcal{V}$ and an assignment m to the variables in $\mathcal{V}(S)$, we denote $m_{|\mathcal{V}}$ to be an assignment to the variables in S such that for each variable $x \in S$, $m_{|\mathcal{V}}(x) = m(\mathcal{V}(x))$.

We denote $s_0 \xrightarrow{BB_0} s_1 \xrightarrow{BB_1} s_2 \ldots \xrightarrow{BB_n} s_n$ to be an execution of a program.

.2 Proofs

We first prove properties of effective dominance sets and governors. We then prove properties of our BMC algorithm.

.2.1 Properties of Effective Dominance Sets and Governors

Lemma .2.1. *Given two basic blocks m and n, if $n \in Edom(m)$, then for each path from m to n, any basic block k along the path is not polluted.*

Proof. Suppose not. Then there is a path from m to n along which there is some k that is

polluted. We consider two cases. Case 1: $m = n$. Then k must be n. Thus n is polluted and is not in $Edom(m)$. Contradiction. Case 2: $m \neq n$. Then $p : m \rightarrow^* k \rightarrow^+ n$. Since k is polluted and n is reachable by k, n is polluted and is not in $Edom(m)$. Contradiction. □

Lemma .2.2. *For any basic block m, $Edom(m)$ is acyclic.*

Proof. Suppose not. Then there is a basic block $n \in Edom(m)$ such that n is the source of a back edge. Hence n is polluted and is not in $Edom(m)$. Contradiction. □

Lemma .2.3. *Let m be a governor. The governed region $GR(m)$ is acyclic.*

Proof. Let $BB1$ and $BB2$ be the successors of m. By Lemma .2.2, $Edom(BB1)$ and $Edom(BB2)$ are acyclic. Moreover, there is not any edge $a \rightarrow b$ where $a \in Edom(BB1)$ and $b \in Edom(BB2)$. Otherwise, we can construct a path $m \rightarrow BB1 \rightarrow^* a \rightarrow b$ which bypasses $BB2$, which indicates that $BB2$ does not dominate b. Similarly, we can prove that there is not any edge $a \rightarrow b$ where $a \in Edom(BB2)$ and $b \in Edom(BB1)$. Thus, $GR(m)$ is acyclic. □

Lemma .2.4. *Let m be a governor. The governed region $GR(m)$ does not have any function calls.*

Proof. Let $BB1$ and $BB2$ be the successors of m. By Lemma .2.1, $Edom(BB1)$ and $Edom(BB2)$ do not have function calls. Since $GR(m) = Edom(BB1) \cup Edom(BB2)$, so does $GR(m)$. □

Lemma .2.5. *A governor m dominates every basic block n in its governed region $GR(m)$.*

Proof. By definition of $GR(m)$, we know that n is either dominated by $BB1$ or $BB2$ where $BB1$ and $BB2$ are the successors of m. Without loss of generality, let us assume that $BB1$ dominates n. Since m is a governor, m dominates $BB1$. Since dominance relation is transitive, m dominates n. □

.2.2 Properties of the BMC Generation Algorithm

Given a governor *gov* and a basic block $BB \in GR(gov)$, note that if g_{BB} is true, then the block formula $Blks[BB]$ encodes the program logic of BB in SSA form, which leads to Lemma .2.6.

Lemma .2.6. *Given a program P and a governor gov, let $\mathcal{V}, \mathcal{V}'$ be the version map before and after the SSA variable renaming for BB. The following two statements hold: (1) If an assignment m with $m(g_{BB}) = true$ is a model of $Blks[BB]$, then $m_{|\mathcal{V}} \xrightarrow{BB} m_{|\mathcal{V}'}$ is an execution of P. (2) If $s \xrightarrow{BB} s'$ is an execution of P, then there is a model m of $Blks[BB]$ such that $m(g_{BB}) = true$, $m_{|\mathcal{V}} = s$, and $m_{|\mathcal{V}'} = s'$.*

Proof. Proof by induction on the number of instructions in BB. □

Given a governor *gov*, we prove that, for any destination $d \in Dests(gov)$, (1) the formula $\phi \wedge g_d$ where $g_d = \bigvee_{e \in Edges[d]} e$ represents all executions from *gov* to d, and (2) the final version of each variable $x \in AccVars(gov)$ in ϕ always represents the value of x when an execution from *gov* to d reaches d.

Lemma .2.7. *Given a governor gov, for any topological ordering T over $GR(gov)$ and any destination $d \in Dests(gov)$, if m is a model of $\phi \wedge g_d$, then (1) we can construct an execution ρ from the governor gov to the destination d, and (2) for each variable $x \in AccVars(gov)$, if x_α is the final version of x in ϕ, then $m(x_\alpha)$ is the value of x when the execution ρ enters the destination d.*

Proof. Since m is a model of $\phi \wedge g_d$, let the set $Taken$ be $\{BB \mid m(g_{BB}) = true\}$, i.e., the set of all basic blocks whose guard g_{BB} is set to true by m. Since g_d is *true*, we know that the guard of a predecessor of d holds, the guard of a predecessor of the predecessor of d holds, and so on. This indicates that there is a path from *gov* to d along which the guards g_{BB} of all basic blocks $BB \in GR(gov)$ are set to *true* by m. Moreover, the guards g_{BB} of all basic blocks $BB \in GR(gov)$ that are not shown along the path are set to *false* by m since the guards of two successors cannot hold at the same time. Hence we know that the set $Taken$ are the intermediate basic blocks of the path from *gov* to d.

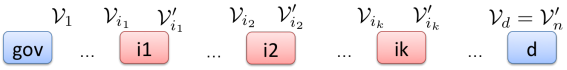


Figure .1: Topological ordering of the governed region

Figure .2: An execution from the governor *gov* to a destination d

As shown in Fig .1, let $BB_1, \ldots, BB_{i_1}, \ldots, BB_{i_2}, \ldots BB_n$ be the sequence of basic blocks in $GR(gov)$ sorted by the topological ordering T such that each $BB_{i_j} \in Taken$ where $1 \leq j \leq k$. Note that $gov, BB_{i_1}, BB_{i_2}, \ldots, BB_{i_k}, d$ is a path. Otherwise, T is not a topological ordering. We now construct an execution along this path. For each $BB_i \in GR(gov)$ where $1 \leq i \leq n$, let $\mathcal{V}_i$ be the version map before BB_i and $\mathcal{V}'_i$ be the one after BB_i. By Lemma .2.6, we know that for each $1 \leq j \leq k$, $m_{|\mathcal{V}_{i_j}} \xrightarrow{BB_{i_j}} m_{|\mathcal{V}'_{i_j}}$ is an execution. Also, since the guards g_{BB} of all basic block $BB \notin Taken$ are set to *false* by the model m, we have

$$m_{|\mathcal{V}_1} = m_{|\mathcal{V}_{i_1}}, \quad \ldots, \quad m_{|\mathcal{V}'_{i_{k-1}}} = m_{|\mathcal{V}_{i_k}}, \quad m_{|\mathcal{V}'_{i_k}} = m_{|\mathcal{V}'_n}$$

Therefore, $\xrightarrow{gov} m_{|\mathcal{V}_{i_1}} \xrightarrow{BB_{i_1}} m_{|\mathcal{V}_{i_2}} \xrightarrow{BB_{i_2}} \ldots \xrightarrow{BB_{i_{k-1}}} m_{|\mathcal{V}_{i_k}} \xrightarrow{BB_{i_k}} m_{|\mathcal{V}'_n} \xrightarrow{d}$ is an execution of the program P. Moreover, since $m_{|\mathcal{V}'_n} \xrightarrow{d}$, the final version of each variable x in the model m represents the value of x when the execution enters the destination d. □

Lemma .2.8. *Given a governor gov, for any topological ordering T over $GR(gov)$ and any destination $d \in Dests(gov)$, if there is an execution from gov to d, then we can construct a model m for the formula $\phi \wedge g_d$.*

Proof. Let $\xrightarrow{gov} s_0 \xrightarrow{BB_{i1}} s_1 \xrightarrow{BB_{i2}} s_2 \ldots \xrightarrow{BB_{i_k}} s_k \xrightarrow{d}$ be an execution from the governor *gov* to a destination d, as shown in Fig .2.

Let $BB_1, BB_2, \ldots, BB_n$ be the sequence of basic blocks in $GR(gov)$ sorted by the topological ordering T. Note that for $2 \leq j \leq k$, $BB_{i_{j-1}}$ must occur before BB_{i_j} along the sequence. Otherwise, T is not a topological ordering. We present this fact in Fig .3.

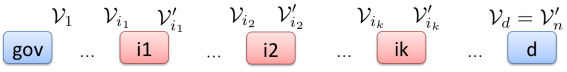


Figure .3: Topological ordering of the governed region

For each $BB_i \in GR(gov)$ where $1 \leq i \leq n$, let $\mathcal{V}_i$ be the version map before BB_i and $\mathcal{V}'_i$ be the one after BB_i. We now construct an assignment m and prove that m is a model of $\phi \wedge g_d$.

Let $Taken$ be the set $\{BB_{i_j} \mid 1 \leq j \leq k\}$. For each basic block $BB \in GR(gov)$, if $BB \in Taken$, then we set $m(g_{BB}) = true$, $m(g_{BB}) = false$ otherwise. For each variable $x \in AccVars(gov)$, we construct the assignment m in four steps:

(1) If $\mathcal{V}_1(x) = x_l$ and $\mathcal{V}_{i_1}(x) = x_h$, then for each x_α with $l \leq \alpha \leq h$, $m(x_\alpha) = s_0(x)$;

(2) For each $j \in [1, k-1]$, if $\mathcal{V}'_{i_j}(x) = x_l$ and $\mathcal{V}_{i_{j+1}}(x) = x_h$, then for each x_α with $l < \alpha \leq h$, $m(x_\alpha) = s_j(x)$,

(3) If $\mathcal{V}'_{i_k}(x) = x_l$ and $\mathcal{V}'_n(x) = x_h$, then for each x_α with $l < \alpha \leq h$, $m(x_\alpha) = s_k(x)$;

(4) For each $j \in [1, k]$, by Lemma .2.6, we know that there is a model m_j of $Blks[BB_{i_j}]$ such that $m_j(g_{BB_{i_j}}) = true$, $m_{j|V_{i_j}} = s_{j-1}$ and $m_{j|V'_{i_j}} = s_j$. If $\mathcal{V}_{i_j}(x) = x_l$ and $\mathcal{V}'_{i_j}(x) = x_h$, then for each x_α with $l < \alpha \leq h$, $m(x_\alpha) = m_j(x)$.

Note that each variable x_α is assigned exactly once in the above construction of m, which means m does not make different values to x_α. Now we show m is indeed a model of $\phi \wedge g_d$. We consider two cases depending on whether a basic block $BB \in GR(gov)$ is in the set $Taken$.

1. Suppose $BB \in Taken$. Then BB is BB_{i_j} for some $j \in [1, k]$. First, $m \models Blks[BB_{i_j}]$ according to Step 4 of the above construction. Secondly, m evaluates $g_{BB_{i_j}}$ to true. Lastly, m evaluates $\bigvee_{c \in Edges[BB_{i_j}]} c$ to true by proving the following cases.

   (a) BB_{i_j} is the left successor of the governor gov. Let $br\ e\ BB_{i_j}\ BB2$ be the terminating instruction of gov. Since $s_0 \models e$ and $m_{|\mathcal{V}_1} = s_0$, we have $m \models \mathcal{V}_1(e)$, and thus $m \models \bigvee_{c \in Edges[BB_{i_j}]} c$.

   (b) BB_{i_j} is the right successor of the governor gov. This case is proved similarly as case (a).

(c) BB_{i_j} is the unique successor of the basic block $BB_{i_{j-1}} \in Taken$. Since $m(g_{BB_{i_{j-1}}}) = \mathit{true}$ and $g_{BB_{i_{j-1}}} \in Edges[BB_{i_j}]$, we have that $m \models \bigvee_{c \in Edges[BB_{i_j}]} c$.

(d) BB_{i_j} is the left successor of $BB_{i_{j-1}} \in Taken$. Let $br\ e\ BB_{i_j}\ BB2$ be the terminating instruction of $BB_{i_{j-1}}$. Since $s_{j-1} \models e$ and $m_{|\mathcal{V}'_{i_{j-1}}} = s_{j-1}$, we have $m \models \mathcal{V}'_{i_{j-1}}(e)$. Moreover, since $m(g_{BB_{i_j-1}}) = \mathit{true}$, then $m \models g_{BB_{i_j-1}} \wedge \mathcal{V}'_{i_{j-1}}(e)$. Hence $m \models \bigvee_{c \in Edges[BB_{i_j}]} c$.

(e) BB_{i_j} is the right successor of $BB_{i_{j-1}} \in Taken$. This case is proved similarly as case (d).

2. Suppose $BB \notin Taken$. First, since $m(g_{BB}) = \mathit{false}$, $Blks[BB]$ are conjunctions of equations of the form $x_\alpha = x_{\alpha-1}$. By Steps (1),(2), and (3), we have $m \models Blks[BB]$. Secondly, we prove that m evaluates $\bigvee_{c \in Edges[BB]} c$ to false by contradiction. Suppose there is $c \in \bigvee_{e \in Edges[BB]} e$ such that $m \models c$. We consider the following cases depending on the form of c.

(a) $c \equiv \mathcal{V}_1(e)$. Then BB is the left successor of the governor gov. Let $br\ e\ BB\ BB_{i_1}$ be the terminating instruction of gov. Since $s_0 \models \neg e$ and $m_{|\mathcal{V}_1} = s_0$, we have $m \models \neg\mathcal{V}_1(e)$, and thus $m \not\models c$. Contradiction.

(b) $c \equiv \neg\mathcal{V}_1(e)$. Then BB is the right successor of the governor gov. This case is proved similarly as case (a).

(c) $c \equiv g_{BB'}$. Then we know that $BB' \in Taken$ and BB is the unique successor of BB'. Since $m(g_{BB'}) = \mathit{true}$, then $m(g_{BB}) = \mathit{true}$. Contradiction.

(d) $c \equiv g_{BB_u} \wedge \mathcal{V}'_u(e)$ for some $u \in [1, n]$. Since $m \models g_{BB_u}$, we know that $BB_u \in Taken$ and BB is the left successor of BB_u. Without loss of generality, let BB_u be BB_{i_j} for some $j \in [1, k]$. Then $\mathcal{V}'_u = \mathcal{V}'_{i_j}$. Let $br\ e\ BB\ BB'$ be the terminating instruction of BB_{i_j}. Note that $s_j \models \neg e$. Since $m_{|\mathcal{V}'_{i_j}} = s_j$, we have $m \models \neg\mathcal{V}'_{i_j}(e)$. Contradiction.

(e) $c \equiv g_{BB_u} \wedge \neg\mathcal{V}'_u(e)$ for some $u \in [1, n]$. Since $m \models g_{BB_u}$, we know that $BB_u \in Taken$ and BB is the right successor of BB_u. This case is proved similarly as case (d).

Now that we have proved for each $BB \in GR(gov)$, $m \models g_{BB} = \bigvee_{c \in Edges[BB]} c$ and $m \models Blks[BB]$. Thus $m \models \phi$. We now prove that $m \models g_d$ where $g_d = \bigvee_{c \in Edges[d]} c$. Suppose that the destination d is the unique successor of BB_{i_k}, then $g_{BB_{i_k}} \in Edges[d]$. Since $m \models g_{BB_{i_k}}$, $m \models g_d$. Suppose that d is the left successor of BB_{i_k}, that is, the terminating instruction of BB_{i_k} is $br\ e\ d\ BB2$. Since $s_k \models e$ and $m_{|\mathcal{V}'_{i_k}} = s_k$, we have $m \models \mathcal{V}'_{i_k}(e)$. Since $g_{BB_{i_k}} \wedge \mathcal{V}'_{i_k}(e) \in Edges[d]$, we have $m \models g_d$. By the similar reasoning, if d is the right successor of BB_{i_k}, we also have $m \models g_d$. Hence $m \models \phi \wedge g_d$. □

Theorem .2.9. *Given a governor gov and a destination $d \in Dests(gov)$, the formula $\phi \wedge g_d$ encodes all executions from gov to d. Moreover, the final version of each variable x in ϕ represents the value of x when an execution from gov to d enters d.*

Proof. Proved by Lemma .2.7 and .2.8. □

[AB12] Alessandro Aldini and Alessandro Bogliolo. "Model checking of trust-based user-centric cooperative networks." In *Proc. 4th International Conference on Advances in Future Internet (AFIN'12)*, pp. 32–41. Citeseer, 2012.

[ABC13] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. "An Orchestrated Survey of Methodologies for Automated Software Test Case Generation." *J. Syst. Softw.*, **86**(8), August 2013.

[AGT08] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. "Demand-driven Compositional Symbolic Execution." In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pp. 367–381, Berlin, Heidelberg, 2008. Springer-Verlag.

[ARC14] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. "Enhancing Symbolic Execution with Veritesting." In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pp. 1083–1094, New York, NY, USA, 2014. ACM.

[ASH16] Pablo Gonzalez-de Aledo, Pablo Sanchez, and Ralf Huuck. *An Approach to Static-Dynamic Software Analysis*, pp. 225–240. Springer International Publishing, Cham, 2016.

[BAJ11] M. Behl, M. Aneja, H. Jain, and R. Mangharam. "EnRoute: An energy router for energy-efficient buildings." In *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pp. 125–126, 2011.

[BCE08] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. "RWset: Attacking Path Explosion in Constraint-based Test Generation." In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, Berlin, Heidelberg, 2008.

[BCG09] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. "Software Model Checking via Large-Block Encoding." In *Proceedings of the 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2009, Austin, TX, November 15-18)*, pp. 25–32, 2009.

[BCM90] J. Burch, E. Clarke, K. McMillan, and D. Dill. "Sequential circuit verification using symbolic model checking." In *27th ACM/IEEE Design Automation Conference*, pp. 46–51, 1990.

[BL05] Mike Barnett and K. Rustan M. Leino. "Weakest-precondition of Unstructured Programs." In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on*

Program Analysis for Software Tools and Engineering, PASTE '05, New York, NY, USA, 2005. ACM.

[BS08] J. Burnim and K. Sen. "Heuristics for Scalable Dynamic Test Generation." In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pp. 443–446, Washington, DC, USA, 2008.

[BZC15] A. Becerra, W. Zeng, M. Chow, and J. Rodriguez-Andina. "Green city: A low-cost Testbed for Distributed Control Algorithms in Smart Grid." In M. Chow A. C. Becerra, W. Zeng and J. Rodriguez-Andina, editors, *Proc. 41st IECON 2015 Annual Conference of the IEEE Industrial Electronics Society*, volume 68, pp. 1948–1953. IEEE Conference, 2015.

[CBR01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. "Bounded Model Checking Using Satisfiability Solving." *Form. Methods Syst. Des.*, **19**(1):7–34, July 2001.

[CCA13] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. "LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems." *ACM Trans. Embed. Comput. Syst.*, **13**(2):24:1–24:27, September 2013.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs." In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pp. 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[CFS09] Satish Chandra, Stephen J. Fink, and Manu Sridharan. "Snugglebug: A Powerful Approach to Weakest Preconditions." In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, New York, NY, USA, 2009. ACM.

[Che07] Steven Cherry. "How to Build A Green City." *IEEE Spectrum*, **44**(6):26–29, January 2007.

[CJW15] Mike Czech, Marie-Christine Jakobs, and Heike Wehrheim. *Just Test What You Cannot Verify!*, pp. 100–114. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: A platform for in-vivo multi-path analysis of software systems." *ACM SIGPLAN Notices*, **46**(3):265–278, 2011.

[CKY03] Edmund Clarke, Daniel Kroening, and Karen Yorav. "Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking." In *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, pp. 368–371, New York, NY, USA, 2003. ACM.

[CLN11] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. "High-Level Synthesis for FPGAs: From Prototyping to Deployment." *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, **30**(4):473–491, April 2011.

[CMN12] Lucas Cordeiro, Jeremy Morse, Denis Nicole, and Bernd Fischer. "Context-Bounded Model Checking with ESBMC 1.17." In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 534–537. Springer Berlin Heidelberg, 2012.

[CRB11] R. Calheiros, R. Rajiv, A. Beloglazov, C. Rose, and R. Buyya. "CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms." *Softw. Pract. Exper.*, **41**(1):23–50, January 2011.

[CS13] Cristian Cadar and Koushik Sen. "Symbolic Execution for Software Testing: Three Decades Later." *Commun. ACM*, **56**(2):82–90, February 2013.

[CSR01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[CZ06] J. Cong and Zhiru Zhang. "An efficient and versatile scheduling algorithm based on SDC formulation." In *2006 43rd ACM/IEEE Design Automation Conference*, pp. 433–438, July 2006.

[CZG13] Ting Chen, Xiao-Song Zhang, Shi-Ze Guo, Hong-Yuan Li, and Yue Wu. "State of the Art: Dynamic Symbolic Execution for Automated Test Generation." *Future Gener. Comput. Syst.*, **29**(7):1758–1773, September 2013.

[DGH15] Przemyslaw Daca, Ashutosh Gupta, and Thomas A. Henzinger. "Abstraction-driven Concolic Testing." *CoRR*, **abs/1511.02615**, 2015.

[Dij97] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[ENS17] W. Ejaz, M. Naeem, A. Shahid, A. Anpalagan, and M. Jo. "Efficient energy management for the internet of things in smart cities." *IEEE Commun. Mag.*, **55**(1):84–91, January 2017.

[FS01] Cormac Flanagan and James B. Saxe. "Avoiding Exponential Explosion: Generating Compact Verification Conditions." In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pp. 193–205, New York, NY, USA, 2001. ACM.

[GHM16] M. Gao, L. He, R. Majumdar, and Z. Wang. "LLSPLAT: Improving Concolic Testing by Bounded Model Checking." In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 127–136, 2016.

[GHW17] M. Gao, L. He, and K. Wang. "Probabilistic Model Checking for Green Energy Router System in Energy Internet." In *2017 IEEE Global Communications Conference (GLOBECOM)*, Dec 2017.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing." In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pp. 213–223, New York, NY, USA, 2005.

[GLM12] Patrice Godefroid, Michael Y. Levin, and David Molnar. "SAGE: Whitebox Fuzzing for Security Testing." *Queue*, **10**(1):20:20–20:27, January 2012.

[God07] Patrice Godefroid. "Compositional Dynamic Test Generation." In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pp. 47–54, New York, NY, USA, 2007. ACM.

[GWH18] M. Gao, K. Wang, and L. He. "Probabilistic Model Checking and Scheduling Implementation of an Energy Router System in Energy Internet for Green Cities." *IEEE Transactions on Industrial Informatics*, **14**(4):1501–1510, April 2018.

[GWY17] Wen-sheng Guo, Yong Wang, Xia Yang, and Min Gao. "Codecomb: Automated Test Case Generation and Defect Detecting for Embedded Software Based on Symbolic Execution." *Journal of Chinese Computer Systems*, **38**(6):1250, 2017.

[HHY15] S. Hambridge, A. Huang, and R. Yu. "Solid State Transformer (SST) as an energy router: Economic dispatch based energy routing strategy." In *2015 IEEE Energy Conversion Congress and Exposition (ECCE)*, pp. 2355–2360, 2015.

[HMP01] G. Haring, R. Marie, R. Puigjaner, and K. Trivedi. "Loss Formulas and their Application to Optimization for Cellular Networks." *IEEE Transaction on Vehicular Technology*, **50**(3):664 –673, May 2001.

[HS11] H. Hildmann and F. Saffre. "Influence of variable supply and load flexibility on Demand-Side Management." In *Proc. 8th International Conference on the European Energy Market (EEM'11)*, pp. 63–68, 2011.

[HSI10] William R. Harris, Sriram Sankaranarayanan, Franjo Ivancic, and Aarti Gupta. "Program analysis via satisfiability modulo path programs." In *POPL 2010*, pp. 71–82. ACM, 2010.

[HTH08] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. "CHStone: A benchmark program suite for practical C-based high-level synthesis." In *2008 IEEE International Symposium on Circuits and Systems*, pp. 1192–1195, May 2008.

[HWC17] J. He, J. Wei, K. Chen, Z. Tang, Y. Zhou, and Y. Zhang. "Multi-tier Fog Computing with Large-scale IoT Data Analytics for Smart Cities." *IEEE Internet of Things Journal*, **PP**(99):1–1, 2017.

[ISO] "ISO New England Data." `https://www.iso-ne.com/`.

[JMN13] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A. Navas. "Boosting Concolic Testing via Interpolation." In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pp. 48–58, New York, NY, USA, 2013. ACM.

[JWW16] H. Jiang, K. Wang, Y. Wang, M. Gao, and Y. Zhang. "Energy big data: A survey." *IEEE Access*, **4**:3844–3861, Aug 2016.

[KCY] Daniel Kroening, Edmund Clarke, and Karen Yorav. "Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking." In *Proceedings of DAC 2003*. ACM Press.

[KJJ09] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley. "Solver technology for system-level to RTL equivalence checking." In *2009 Design, Automation Test in Europe Conference Exhibition*, pp. 196–201, April 2009.

[KKB12] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. "Efficient State Merging in Symbolic Execution." *SIGPLAN Not.*, **47**(6):193–204, June 2012.

[KLG08] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. "Validating High-Level Synthesis." In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, pp. 459–472, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[KNP07] Marta Kwiatkowska, Gethin Norman, and David Parker. "Stochastic Model Checking." In *Proceedings of the 7th International Conference on Formal Methods for Performance Evaluation*, SFM'07, pp. 220–270, Berlin, Heidelberg, 2007. Springer-Verlag.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. "PRISM 4.0: Verification of Probabilistic Real-time Systems." In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806, pp. 585–591. Springer, 2011.

[KSM08] C. Karfa, D. Sarkar, C. Mandal, and P. Kumar. "An Equivalence-Checking Method for Scheduling Verification in High-Level Synthesis." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **27**(3):556–569, March 2008.

[KST13] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C.M. Wintersteiger. "Loop summarization using state and transition invariants." *Formal Methods in System Design*, **42**(3):221–261, 2013.

[LQL12] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. "A Solver for Reachability Modulo Theories." In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pp. 427–443, Berlin, Heidelberg, 2012. Springer-Verlag.

[LSH11] C. H. Lee, C. H. Shih, J. D. Huang, and J. Y. Jou. "Equivalence checking of scheduling with speculative code transformations in high-level synthesis." In *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)*, pp. 497–502, Jan 2011.

[LSW13] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. "Steering Symbolic Execution to Less Traveled Paths." In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pp. 19–32, New York, NY, USA, 2013. ACM.

[MFC09] A. Mathur, M. Fujita, E. Clarke, and P. Urard. "Functional Equivalence Verification Tools in High-Level Synthesis Flows." *IEEE Design Test of Computers*, **26**(4):88–95, July 2009.

[MFS12] Florian Merz, Stephan Falke, and Carsten Sinz. "LLBMC: Bounded Model Checking of C and C++; Programs Using a Compiler IR." In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE'12, pp. 146–161, Berlin, Heidelberg, 2012. Springer-Verlag.

[MS07] Rupak Majumdar and Koushik Sen. "Hybrid Concolic Testing." In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pp. 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[MS09] Grant Martin and Gary Smith. "High-level synthesis: Past, present, and future." *IEEE Design & Test of Computers*, **26**(4):18–25, 2009.

[MSC13] C. Marquez, M. Strum, and W. Chau. "Formal equivalence checking between high-level and RTL hardware designs." *2013 14th Latin American Test Workshop - LATW*, **00**:1–6, 2013.

[MTG00] M. Moreno, F. Terroso, A. Gonzalez, M. Valdes, A. Skarmeta, M. Zamora, and V. Chang. "Applicability of big data techniques to smart cities deployments." *IEEE Trans. Industrial Informatics*, **13**(2):800–809, apr 2000.

[MTK16] Rajdeep Mukherjee, Michael Tautschnig, and Daniel Kroening. "v2c – A Verilog to C Translator." In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 580–586, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[MZK17] J. Miao, N. Zhang, C. Kang, J. Wang, Y. Wang, and Q. Xia. "Steady-state Power Flow Model of Energy Router Embedded AC Network and Its Application in Optimizing Power System Operation." *IEEE Transactions on Smart Grid*, **PP**(99):1–1, 2017.

[NMF06] T. Nishihara, T. Matsumoto, and M. Fujita. "Equivalence Checking with Rule-Based Equivalence Propagation and High-Level Synthesis." In *2006 IEEE International High Level Design Validation and Test Workshop*, pp. 162–169, Nov 2006.

[OF15] O. Ojala and T. Ferm. "Building green city with green choices in traffic." In O. Ojala and T. Ferm, editors, *Proc. eChallenges e-2015 Conference*, volume 68, pp. 1–8. IEEE Conference, 2015.

[QM14] Xiaoke Qin and Prabhat Mishra. "Scalable test generation by interleaving concrete and symbolic execution." In *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on*, pp. 104–109. IEEE, 2014.

[QP99] Qinru Qiu and M. Pedram. "Dynamic power management based on continuous-time Markov decision processes." In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pp. 555–561, 1999.

[SH10] Raul Santelices and Mary Jean Harrold. "Exploiting Program Dependencies for Scalable Multiple-path Symbolic Execution." In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, New York, USA, 2010.

[SK14] Hyunmin Seo and Sunghun Kim. "How We Get There: A Context-guided Search Strategy in Concolic Testing." In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, New York, NY, USA, 2014. ACM.

[SMA05] 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, pp. 263–272, New York, NY, USA, 2005. ACM.

[SOG10] A. Sanchez-Squella, R. Ortega, R. Grino, and S. Malo. "Dynamic Energy Router." *IEEE Control Systems*, **30**(6):72–80, Nov 2010.

[SPF14] T. Su, G. Pu, B. Fang, J. He, J. Yan, S. Jiang, and J. Zhao. "Automated Coverage-Driven Test Data Generation Using Dynamic Symbolic Execution." In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pp. 98–107, June 2014.

[SVC] SVCOMP15. "Competition on Software Verification." `https://github.com/dbeyer/sv-benchmarks/tree/master/c/`.

[Ueh16] Tadahiro Uehara. "Exhaustive Test-case Generation using Symbolic Execution." 2016.

[USK14] J. Urdahl, D. Stoffel, and W. Kunz. "Path Predicate Abstraction for Sound System-Level Models of RT-Level Circuit Designs." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **33**(2):291–304, Feb 2014.

[WHL17] K. Wang, X. Hu, H. Li, P. Li, D. Zeng, and S. Guo. "A Survey on Energy Internet Communications for Sustainability." *IEEE Transactions on Sustainable Computing*, **PP**(99):1–1, 2017.

[WLF17] K. Wang, H. Li, Y. Feng, and G. Tian. "Big Data Analytics for System Stability Evaluation Strategy in the Energy Internet." *IEEE Transactions on Industrial Informatics*, **13**(4):1969–1978, Aug 2017.

[Wol] Clifford Wolf. "Yosys Open SYnthesis Suite." `http://www.clifford.at/yosys/`.

[WWH17] K. Wang, Y. Wang, X. Hu, Y. Sun, D. J. Deng, A. Vinel, and Y. Zhang. "Wireless Big Data Computing in Smart Grid." *IEEE Wireless Communications*, **24**(2):58–64, April 2017.

[WYY17] K. Wang, J. Yu, Y. Yu, Y. Qian, D. Zeng, S. Guo, Y. Xiang, and J. Wu. "A Survey on Energy Internet: Architecture, Approach, and Emerging Technologies." *IEEE Systems Journal*, **PP**(99):1–14, Jan 2017.

[XDS15] L. Xin, Z. Dong, Y. Sun, J. Hou, and J. Liu. "Design and application of energy router to realise Energy Internet." In *10th International Conference on Advances in Power System Control, Operation Management (APSCOM 2015)*, pp. 1–6, 2015.

[XTH09] Tao Xie, N. Tillmann, J. de Halleux, and W. Schulte. "Fitness-guided path exploration in dynamic symbolic execution." In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pp. 359–368, June 2009.

[XZW11] Yi Xu, Jianhua Zhang, Wenye Wang, A. Juneja, and S. Bhattacharya. "Energy router: Architectures and functionalities toward Energy Internet." In *2011 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pp. 31–36, 2011.

[YZJ16] P. Yi, T. Zhu, B. Jiang, R. Jin, and B. Wang. "Deploying Energy Routers in an Energy Internet Based on Electric Vehicles." *IEEE Transactions on Vehicular Technology*, **65**(6):4714–4725, May 2016.

[ZYS16] W. Zhong, R. Yu, X. Shengli, Y. Zhang, and D. K. Y. Yau. "On Stability and Robustness of Demand Response in V2G Mobile Energy Networks." *IEEE Transactions on Smart Grid*, **PP**(99):1–1, 2016.

[ZYX11] Y. Zhang, R. Yu, S. Xie, W. Yao, Y. Xiao, and M. Guizani. "Home M2M networks: Architectures, standards, and QoS improvement." *IEEE Communications Magazine*, **49**(4):44–52, April 2011.

[ZYX16] W. Zhong, R. Yu, S. Xie, Y. Zhang, and D. Tsang. "Software Defined Networking for Flexible and Green Energy Internet." *IEEE Communications Magazine*, **54**(12):68–75, Dec 2016.