DISTRIBUTED SYMBOLIC EXECUTION FOR PROPERTY CHECKING

by

Junye Wen

A thesis submitted to the Graduate College of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
May 2017

Committee Members:

Guowei Yang, Chair

Xiao Chen

Rodion Podorozhny

COPYRIGHT

by

Junye Wen

2017

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

This work is protected by the Copyright Laws of the United States (Public Law 94-553, section 107). Consistent with fair use as defined in the Copyright Laws, brief quotations from this material are allowed with proper acknowledgement. Use of this material for financial gain without the author's express written permission is not allowed.

Duplication Permission

As the copyright holder of this work I, Junye Wen, authorize duplication of this work, in whole or in part, for educational or scholarly purposes only.

DEDICATION

Dedicated to my family, who have always loved and supported me unconditionally and whose good examples have taught me and given me confidence to achieve the best I can.

ACKNOWLEDGEMENTS

I deeply appreciate my supervisor Dr. Guowei Yang, who guides me during my graduate program. The past two and half years has been a great pleasure to me, with his valuable guidance,financial support, encouragement, and patience. This thesis would not have been possible without his guidance and persistent help. I am also grateful to my committee members, Dr. Rodion Podorozhny and Dr. Xiao Chen, for constant support and insightful comments in this thesis.
This work was funded in part by the National Science Foundation under Grant No. CCF-1464123.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS v

LIST OF TABLES viii

LIST OF FIGURES ix

LIST OF ABBREVIATIONS x

ABSTRACT xi

CHAPTER

I. INTRODUCTION 1

Motivation 1

Goals of Research 2

Research Methodology 2

Outline 3

II. BACKGROUND 4

Symbolic Execution 4

Symbolic Pathfinder 7

III. APPROACH 9

Foundation 9

Distributed Assertion Checking 10

Algorithms 12

Implementation 15

IV. EVALUATION 19

Tool Support 19

Assumptions 20
Artifacts 21
Metrics and Research Questions 22
Results and Analysis 23
Discussions 28

V. RELATED WORK 33

VI. FUTURE WORK 35

VII. CONCLUSION 37

APPENDIX SECTION 38

REFERENCES 41

LIST OF TABLES

Table **Page**

IV.1 Results of AssertionTest 25
IV.2 Results of WBS 26
IV.3 Results of trityp 26
IV.4 Results of Apollo 27
IV.5 Time cost when workers running on same/different cores 30

LIST OF FIGURES

Figure **Page**

II.1 Symbolic Execution Sample: (a) Code that swaps two integers, (b) the corresponding symbolic execution tree, and (c) test data and path constraints corresponding to different program paths. 5

II.2 Illustration of JPF Core 7

II.3 JPF Listeners 8

III.1 Example Trie of Distributed Exploration: (a) example trie explored by the first stage; (b) paths distributed to multiple workers for parallel assertion checking. 11

III.2 The work flow of technique 12

III.3 Algorithm for the First Stage 13

III.4 Algorithm for the Second Stage 14

III.5 A Sample Bytecode Corresponding to Assertion 16

IV.1 Source Code of AssertionTest 21

IV.2 Example trie leading to worse performance 28

LIST OF ABBREVIATIONS

Abbreviation Description

CG Choice Generator

JPF Java Pathfinder

JVM Java Virtual Machine

Memoise Memoized Symbolic Execution

PC Path Condition

SPF Symbolic Pathfinder

TACC Texas Advanced Computing Center

VM Virtual Machine

WBS Wheel Brake System

ABSTRACT

Symbolic execution is a powerful technique for checking properties in the form of assertions. It can systematically explore the program's state space to find paths to assertion violations, and provide users with a counterexample for each violation by solving the corresponding path condition using the underlying constraint solver. However, conventional symbolic execution is configured either to explore all the state space to find all possible assertion violations, or to stop when it finds the first assertion violation. In the former case, one symbolic execution run could take a long time before giving any result, and users have to wait for the whole execution time before they could take any action to deal with the potential problems in their code. In the latter case, users need to run symbolic execution for multiple times to check all assertion violations.

In this thesis, we develop a novel technique to check assertions in parallel using symbolic execution. Our technique improves the efficiency of assertion checking using symbolic execution and gives users earlier results. This technique is in two stages: in the first stage, a worker is launched to find feasible paths to the checked assertions, then in the second stage, multiple workers are launched to check assertions on these paths in parallel simultaneously, each worker focusing on one of these feasible paths. We evaluate our technique using experiments with four Java subjects, and the experimental results show its effectiveness and efficiency.

I. INTRODUCTION

Motivation

Annotating functional correctness properties of code, e.g., using assertions (Clarke and Rosenblum, 2006) or executable contracts (Leavens et al., 2005; Meyer et al., 1987), enables automated conformance checking of program behaviors to expected properties, and is widely used for finding bugs (Corbett et al., 2000; Godefroid, 1997). However, effectively utilizing such properties in practice is complicated, largely due to the complexity of specifying and maintaining them as well as the high computational cost of checking them. In our research, we are focusing on checking properties which are assertions in source code.

Symbolic execution (King, 1976a) uses symbolic values, instead of concrete values, as program inputs, and represents the values of program variables as symbolic expressions of those inputs. By solving the corresponding path condition using the underlying constraint solver, it can check all feasible paths of given subjects within depth boundary set by users. It is suitable as a powerful technique for checking assertions. Symbolic execution can systematically explore the program's state space to find paths to assertion violations, and provide users with a counterexample for each violation by solving the corresponding path condition using the underlying constraint solver.

However, suffering from problems of path explosion, path divergence, and expensive constraint solving, checking assertions using conventional symbolic execution can be expensive. Usually, symbolic execution is configured either to explore all the state space to find all possible assertion violations, or to stop when it finds the first assertion violation. In the former case, one symbolic execution run could take a long time before giving any result, and users have to wait for the whole execution time before they could take any action to deal with

the potential problems in their code. In the latter case, users need to run symbolic execution for multiple times to check all assertion violations.

Goals of Research

Our research is aiming at finding assertion violations more efficiently in two aspects: first, one single symbolic execution run could find as many violations as possible, so users do not need to run a potentially long-time symbolic execution run sequentially for multiple times; second, if a potential violation is detected, users can get a report about it at the earliest time possible so they can start working on their code at once, even while the exploration is still on going.
One possible way to archive this goal is to distribute checking to several workers in parallel, while each worker is solving a much simpler problem. As a rule, assertions are usually required to be side effect free, which means the checking result of one assertion is not dependent on the checking result of another assertion. Thus, it guarantees that distribution of assertion checking would not change the final result.
In our research, we would like to find a way to analyze and divide the given subject into smaller working packages, and then distribute them to multiple working units. We would like to design and implement a assertion checking approach that is more efficient than using conventional symbolic execution technique.

Research Methodology

Although several assertion checking techniques using symbolic execution have been developed (Khurshid et al., 2003; Deng et al., 2006; Zhang et al., 2015), most of them are using serial algorithms. In our research, we use parallel algorithms to solve the problem to improve the efficiency. Our approach first launches a worker to find all feasible paths to assertions as target paths. For each target path, a worker is launched to check the assertion along that path.

Memoized Symbolic Execution (Memoise) is used to quickly traverse the target path, check the corresponding assertion, and provide users a report about the result.

We use an open source framework for symbolic execution, Symbolic Pathfinder (SPF) (Păsăreanu and Rungta, 2010), as our platform to implement the approach. Also, to examine the efficiency of our technique, we run several expriments for evaluation. We used a handmade subject as well as real world subjects to evaluate our technique. The results show that our technique can find all assertion violations reported by conventional symbolic execution, while using less time and providing users earlier reports of assertion checking.

Outline

This thesis is structured as follows. In Chapter 2, we introduce the background techniques, including symbolic execution and JPF framework. In Chapter 3, we describe the algorithms and implementation of our approach. Chapter 4 evaluates our technique using a set of experiments. In Chapter 5 discusses related work. Chapter 6 discusses future work to further improve our technique, and Chapter 7 concludes this thesis.

II. BACKGROUND

In this chapter, we introduce the concept of symbolic execution, which is the background technique of our approach. Also, we briefly introduce SPF, a widely used symbolic execution framework for Java.

Symbolic Execution

Symbolic execution (King, 1976a) uses symbolic values, instead of concrete values, as program inputs, and represents the values of program variables as symbolic expressions of those inputs. At any point during symbolic execution, the state of a symbolically executed program includes the symbolic values of program variables at that point, a path condition on the symbolic values to reach that point, and a program counter. The Path Condition (PC) is a Boolean formula over the symbolic inputs, which is an accumulation of the constraints that the inputs must satisfy for an execution to follow that path. At each branch point during symbolic execution, the PC is updated with constraints on the inputs such that (1) if the PC becomes unsatisfiable, the corresponding program path is infeasible, and symbolic execution does not continue further along that path and (2) if the PC is satisfiable, any solution of the PC is a program input that executes the corresponding path. The program counter identifies the next statement to be executed.

To illustrate, consider the code fragment in Figure 1(a) that swaps the values of integer variables x and y, when the initial value of x is greater than the initial value of y; we reference statements in the figure by their line numbers. Figure 1(b) shows the symbolic execution tree for the code fragment. A symbolic execution tree is a compact representation of the execution paths followed during the symbolic execution of a program. In the tree, nodes represent program states, and edges represent transitions between states. The numbers shown at

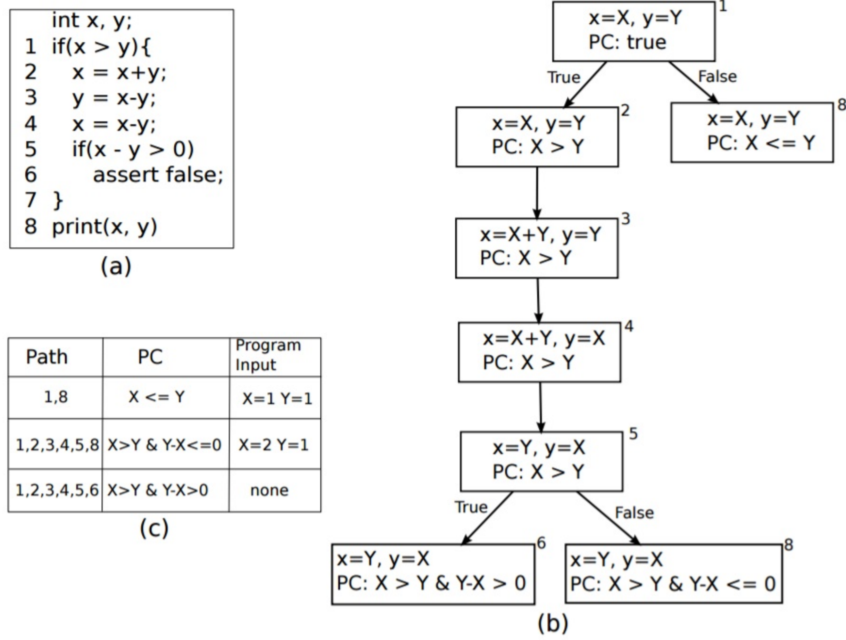

Figure II.1: Symbolic Execution Sample: (a) Code that swaps two integers, (b) the corresponding symbolic execution tree, and (c) test data and path constraints corresponding to different program paths.

the upper right corners of nodes represent values of program counters. Before execution of statement 1, the PC is initialized to true because statement 1 is executed for any program input, and x and y are given symbolic values X and Y, respectively. The PC is updated appropriately after execution of if statements 1 and 5. The table in Figure 1(c) shows the PC's and their solutions (if they exist) that correspond to three program paths through the code fragment. For example, the PC of path (1,2,3,4,5,8) is $X > Y \& Y - X <= 0$. Thus, a program input that causes the program to take that path is obtained by solving the PC. One such program input is $X = 2$, $Y = 1$. For another example, the PC of path (1,2,3,4,5,6) is an unsatisfiable constraint $X > Y \& Y - X > 0$, which means that there is no program input for which the program will take that (infeasible) path. Symbolic execution offers the advantage that one symbolic execution may represent a large, usually infinite, class of normal executions. This can be used to great advantage in program analyzing, testing and debugging. However, to build a symbolic execution system that is both efficient and automatic, three fundamental problems of the technique must be addressed: path explosion, path

divergence and complex constraint solving (Anand et al., 2013).

Symbolic execution is a powerful technique for checking assertions, especially for checking assertions in the form of assertions. It can systematically explore the program's state space to find paths to assertion violations, and provide users with a counterexample for each violation by solving the corresponding path condition using the underlying constraint solver. However, conventional symbolic execution is also suffering from some problems(Anand et al., 2013).

- `Path explosion.` It is difficult to symbolically execute a significantly large subset of all program paths because (1) most real world software have an extremely large number of paths, and (2) symbolic execution of each program path can incur high computational overhead. Thus, in reasonable time, only a small subset of all paths can be symbolically executed. The goal of discovering a large number of feasible program paths is further jeopardized because the typical ratio of the number of infeasible paths to the number of feasible paths is high (Ngo and Tan, 2007). This problem needs to be addressed for efficiency of a symbolic execution system.

- `Complex constraints.` It may not always be possible to solve path constraints because solving the general class of constraints is undecidable. Thus, it is possible that the computed path constraints become too complex (e.g., constraints involving nonlinear operations such as multiplication and division and mathematical functions such as sin and log), and thus, cannot be solved using available constraint solvers. The inability to solve path constraints reduces the number of distinct feasible paths a symbolic execution system can discover.

As for checking assertions, symbolic execution is usually configured either to explore all the state space to find all possible assertion violations, or to stop when it finds the first assertion violation. In the former case, due to the three basic problems mentioned above, one symbolic execution run could take a long

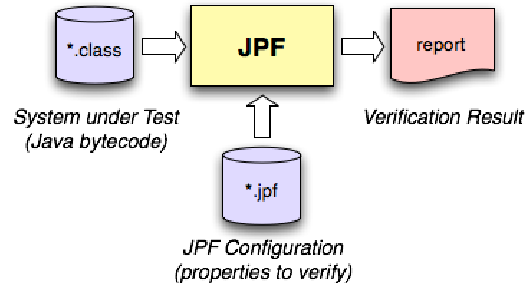

Figure II.2: Illustration of JPF Core

time before giving any result, and users have to wait for the whole execution time before they could take any action to deal with the potential problems in their code. In the latter case, users need to run symbolic execution for multiple times to check all assertion violations.

Symbolic Pathfinder

SPF is such a Java framework specially designed for symbolic execution. SPF combines symbolic execution with model checking and constraint solving for test case generation. In this tool, programs are executed on symbolic inputs instead of concrete values. Values of variables are represented as constraints over the symbolic variables, generated from analysis of the code structure. These constraints are then solved to generate test inputs. Essentially SPF performs symbolic execution for Java programs at the bytecode level.

SPF uses the analysis engine of Java Pathfinder (JPF). The core of JPF (Havelund and Pressburger, 2000) is a Virtual Machine (VM) for Java bytecode which is generally used as model checker. A brief illustration of its work-flow is shown in Figure II.2. It is used to find defects in given Java programs, with the assertions to check for given as input. JPF gets back to user with a report that says if the assertions hold and/or which verification artifacts have been created by JPF for further analysis (like test cases).

The Java Virtual Machine (JVM) of JPF is the Java specific state generator. By executing Java bytecode instructions, the JVM generates state representations

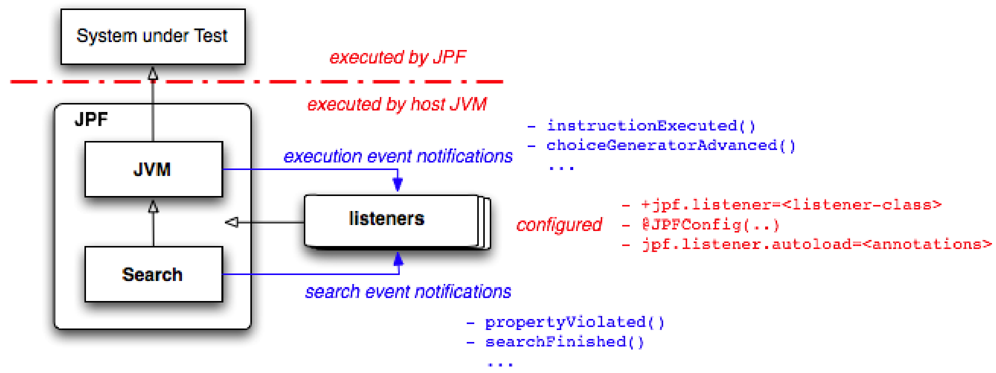

Figure II.3: JPF Listeners

that can be checked, queried, stored and restored. JPF provided interfaces for developer to control these behaviors. As shown in Figure II.3, certain events (e.g. instruction executed, choice generator registered, state backtracked, etc.) will trigger a series of Java interfaces (listeners), and developer could control the flow of execution by implement their own listener. For more information about listeners in JPF, please refer Appendix B.

III. APPROACH

In this chapter, we first introduce the foundation of our technique, and then describe the algorithms and implementation.

Foundation

This section provides a brief introduction of *Memoise*, which serves as the foundation of the development of our technique.

The key insight in Memoise is that applying symbolic execution often requires several successive runs of the technique on largely similar underlying problem instances. Memoise leverages the similarities to reduce the total cost of applying the technique by maintaining and updating the *computations* involved in a symbolic execution run. It reduces both the number of paths to explore by pruning the path exploration as well as the cost of constraint solving by re-using previously computed constraint solving results.

Specifically, Memoise uses a trie (Fredkin, 1960; Willard, 1984)—an efficient tree-based data structure—for a compact representation of the paths visited during a symbolic execution run. Essentially, the trie records the *choices* taken when exploring different paths, together with bookkeeping information that maps each trie node to the corresponding condition in the code. Maintenance of the trie during successive runs allows *re-use* of previous computation results of symbolic execution without the need for re-computing as is traditionally done. Constraint solving is turned off for previously explored paths and the search is guided by the choices recorded in the trie. An initial run of Memoise performs standard symbolic execution as well as builds the trie on-the-fly and saves it on the disk for future re-use. To facilitate future runs of symbolic execution, a subset of the leaf nodes in a trie is partitioned into a set of *boundary* nodes, which are leaf nodes because of the chosen depth bound, and a set of

unsatisfiable nodes, which are leaf nodes due to unsatisfiable path conditions. Based on the results cached in the trie, Memoise enables efficient re-execution, which is guided by the trie using algorithms that are specialized for the particular analysis that is performed.

In our approach, Memoise is used to improve the efficiency of our technique.

Distributed Assertion Checking

Our technique aims at finding assertion violations more efficiently in two aspects: first, one single symbolic execution run could find as many violations as possible, so users do not need to run a potentially long-time symbolic execution run for multiple times; second, if a potential violation is detected, users can get a report about it at the earliest time possible so they can start working on their code at once, even while the exploration is still on going.

One possible way to archive this goal is to distribute checking to several workers in parallel, while each worker is solving a much simpler problem. As a rule, assertions are usually required to be side effect free, which means the checking result of one assertion is not dependent on the checking result of another assertion. Thus, it guarantees that distribution of assertion checking would not change the final result. Moreover, as a typical embarrassingly parallel problem, the partial result of each worker does not need to be reduced into one overall result. In other words, once the exploration is started, there is very limited amount of data transferring among workers, and complex synchronization mechanism between workers is not necessary. Thus, the partial results could be provided to users as soon as they come out, so users could use them to immediately check the possible problems in the assertion or in the code.

A key challenge in distributing an analysis run is to minimize the communication cost among different workers that may be on different machines. A distribution technique that creates independent workloads is highly desirable. We introduce a novel parallel assertion checking technique based on staged symbolic execution,

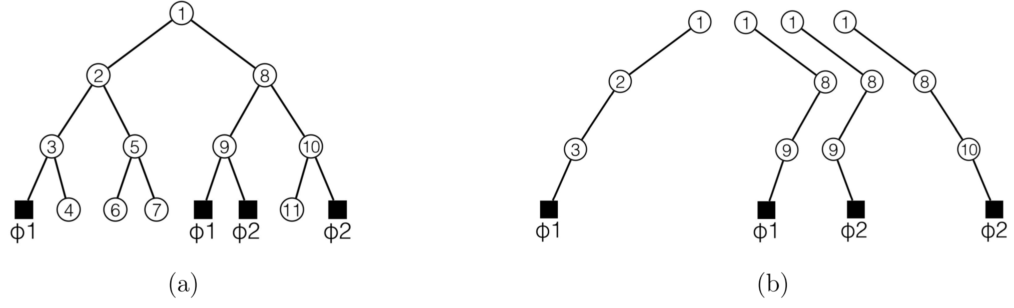

Figure III.1: Example Trie of Distributed Exploration: (a) example trie explored by the first stage; (b) paths distributed to multiple workers for parallel assertion checking.

so that the trie created in the first stage is used to define static workloads for parallel workers:

- In the first stage, we perform symbolic execution to explore all program behaviors and find all feasible paths that lead to the checked assertions. In this stage, we filter out all paths which are irrelevant to assertion checking, and save target paths which are used in the second stage with a focus on checking assertions.
- In the second stage, multiple workers are launched in parallel to explore the state space for checking assertions guided by the target paths provided by the first stage. Each worker is guided by one target path using Memoise, and explores the sub-state space corresponding to the reached assertion. All the workers are run in parallel, and individually output the result to users once any of them finishes its job.

Figure III.1 shows the two stages symbolic execution on a small subject with two assertions. In the first stage, a symbolic execution run is launched, where a state space trie is built as the exploration is proceeded and four target paths, which are feasible and lead to the checked assertions, are found. After gathering all needed information, in the second stage, four workers are launched, each focusing checking a assertion along one target path, while avoiding the exploration of

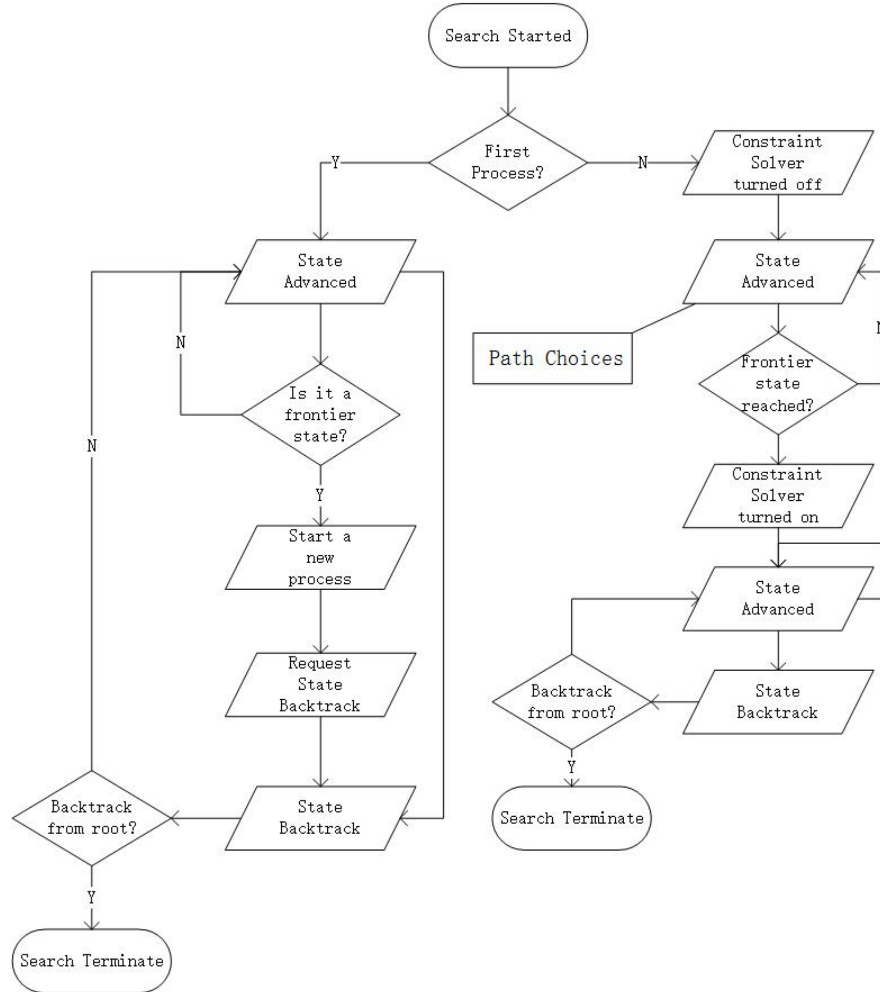

Figure III.2: The work flow of technique

irrelevant states and efficiently exploring the four target paths by turning off constraint solving.

In practice, we do *not* wait for the symbolic execution to finish the first stage to start the second one. To be more specifically to our implementation, as soon as first worker reached a frontier state, it launches another worker immediately before start looking for other feasible paths to assertion checking. In other words, if we generally looking the whole exploration, both stages are run simultaneously by multiple workers.

Algorithms

Description of algorithms for both stages are shown in pseudo-code in Figure III.3 and Figure III.4. To further help understanding, work flow of our approach

Algorithm 1 Algorithm for the First Stage

```
Input: Program P, List of assertion L, Depth Boundary D
Output: Report of symbolic execution R
1:  Path Choice currentPathChoice ← null
2:  PathChoices List pathChoices ← empty
3:  Current Depth depth ← 0
4:  State s ← root state of P
5:  Choice Generator gc ← null
6:  while ( d < D ) do
7:      if s advanced then
8:          pathChoices ← pathChoices + currentPathChoice
9:          depth ← depth +1
10:     if s backtrack then
11:         if s is root state then
12:             break
13:         else
14:             pathChoices ← pathChoices– last choice in pathChoices
15:             depth ← depth -1
16:     if choice generator registered then
17:         Instruction i ← last instruction executed
18:         if i is in L then
19:             start new worker W (input: P, D, pathChoices)
20:             symbolic execution backtrack
21:         else
22:             gc ← choice generator registered
23:     if choices in choice generator advanced then
24:         currentPathChoice ← index of processing choice in gc
25: generate R
26: return R
```

Figure III.3: Algorithm for the First Stage

is given in Figure III.2.

Figure III.3 shows the algorithm for the first stage. Worker launched in this stage applies a normal symbolic execution, aiming to find assertions on feasible paths. During the exploration, this worker collects the choices processed as it reaches every new state, and maintains a list of the choices. The list is updated as the state advances or backtracks. Meanwhile, this worker monitors the instructions executed and checks if they are in the list of assertion instructions given as input. If the instruction is in the list, which eventually triggered a new

Algorithm 2 Algorithm for the Second Stage

Input: Program *P*, List of assertion *L*, Depth Boundary *D*, PathChoices List *pathChoices*

Output: Report of symbolic execution *R*

```
1:  Constraint solver turned off
2:  Current Depth depth ← 0
3:  State s ← root state of P
4:  Frontier State fs ← null;
5:  Choice Generator gc ← null
6:  while ( d < length of pathChoices ) do
7:      if choice generator registered then
8:          gc ← choice generator registered
9:          choice c ← choice at index d in pathchoices
10:         gc advance to c
11:         s advance
12: Constraint solver turned on
13: fs ← s
14: while ( d < D ) do
15:     /*conventional symbolic execution*/
16:     if s backtrack then
17:         if s == fs then
18:         break;
19: generate R
20: return R
```

Figure III.4: Algorithm for the Second Stage

choice generator corresponded to it registered, we know that the current state is a frontier state that stands for assertion checking. After reaching a frontier state, this worker launches a new worker, and assign it with currnt path by sending the path choices that leading to this state from the root of trie. Then, the current state stops searching the current path, and backtracks to explore other part of the state space.

The algorithm for the second stage are shown in Figure III.4. Workers in this stage follow the given path choices to quickly traverse through the state space towards the assigned frontier state. PC solver is turned off, for the path constraints on this path has already been solved and we are sure all states are feasible. Upon reaching the frontier state, exploration first turns on the PC solver to check feasibility of the corresponding assertion, and gives a counter-example

that may cause violation is possible. Then, it logically uses the frontier state as the root state of deeper part of the trie. It continues traversing the state space beneath the frontier state, using normal symbolic execution strategy. Search is stopped when it backtracks from the frontier state, for the state space rooted at the frontier state is fully explored. Then, a report is generated regarding the validity of all the checked assertions this worker has encountered. Since all workers are run in parallel, earlier feedback could be expected in comparison to the conventional symbolic execution, which only gives an overall report after exploring all feasible states in configuration where all errors need to be explored. Currently, our technique does not check the reach-ability of assertions statically. In other words, if there are no assertions at all in the program, the first worker will explore the whole state space in the same way as the default symbolic execution, while monitoring the execution of instructions. Even in this case, since the list of assertion instruction is empty, the cost should be similar as conventional symbolic execution, which is acceptable. However, if we could leverage some static analysis technique to avoid paths that are deemed to reach no assertions, which is left for future work, more reduction in analysis cost could be expected.

Implementation

Since JPF/SPF is running on Java bytecode rather than source code, our first step of implementation is to mark out all the instructions representing an assertion in bytecode. After compiled, an assertion is transformed in a series of bytecodes that equals to if-branch throwing a certain type of exception. A sample code fragment in Figure III.5.

A typical assertion is in a format of three parts:

- Head part.In head part, system variable #2 is loaded. This is the variable indicates whether assertion is turned on in visual machine. If it is not, the following instruction "ifne xx" guides the execution to skip all assertion

```
16: getstatic     #2  // Field $assertionsDisabled:Z
19: ifne          34
22: iload_0
23: ifgt          34
26: new           #3  // class java/lang/AssertionError
29: dup
30: invokespecial #4 // Method java/lang/AssertionError."<init>":()V
33: athrow
34: ...
```

Figure III.5: A Sample Bytecode Corresponding to Assertion

instructions. Line 16 and 19 in Figure III.5 is corresponding to head part.

- Body part. If assertion function is on, instructions represent the actual condition of the assertion will be executed. This part could be a serial of if-branch bytecodes or method calls, and guild to the assertion error part if condition is not satisfied. Noted that body part could be very simple, as Line 22 and 23 in Figure III.5, or could be very long and complicate according to the assertion condition.

- Foot part. This part represents that when an assertion is violated. As shown in Line 26 to 33 in Figure III.5, a typical assertion exception is thrown. Every assertion code ends with these bytecode instructions.

Ideally, instruction "getstatic #2" can be used as the indicator of the start of an assertion. However, we find out this instruction is difficult to catch in SPF when there are several assertions in the code. In most cases, this part of the code is only loaded and executed at the very beginning of execution and the first assertion actually reached. In other words, once JVM had confirmed that assertion function was turned on, it will not check this configuration again in middle of an execution run. In case of JPF, later head parts of assertion bytecodes are totally ignore and cannot be captured by any listeners. In order to make sure we can catch every assertions, we used an alternate solution that

catch the if-instructions in body part instead. When SPF reaches such instructions, it treats them as normal if-branches and register a Choice Generator (CG) for it, which could be captured by listeners in JPF.
Currently, we use Javap -c command to generate and manually input the instruction ID into the program.We noticed that some tools, i.e. Apache Commons BCEL (https://commons.apache.org/proper/commons-bcel/) could be useful to help us automatically conduct bytecode analysis. But currently, we are checking code manually to make sure it is accurate.
After gathering all information of the checked assertions, we start the symbolic execution run using it to control the flow of exploration. We implemented our technique using several listeners provided by SPF. The main listeners we used are as follows:

- `Choice Generator listeners.` Choice Generators is a mechanism JPF used as a set of possible transitions from a current state to a new state. For example, an if-instruction is explained with a choice generator registered with two choices, each of which leading to a different state as the condition Boolean is true or false. Each of these choices is stored with an index, and symbolic execution uses Depth First Search strategy by default to check every choice one by one, thus explore all corresponding states. JPF has provided several listeners answering to different behaviors of Choice Generator. In our implementation, we mainly used two of these listeners: (1) choiceGeneratorRegistered, which is triggered as a choice generator is initialized with choices. Since assertions are in form of if-branches, each of these assertions eventually have a choice generator standing for it. Thus, when a choice generator is registered, we can check if we need to skip some choices that represents states not on a critical path. For the first worker, if our current state is a frontier state, this event also indicates the end of the state, and giving us a control point to start a new worker and backtrack to parent states. (2)choiceGeneratorAdvanced, which is called every time JPF

is going to get a worker a choice of the Choice Generator. To perform Memoise, our approach need to transfer the guide to later workers about the path leading to a frontier state. Since the path can be indicated as which choice we processed at each choice generator, we dynamically maintained a string which is indexes of all choices leading to the current state we are exploring. If it is a frontier state, we send this string, or "Path Choices", to the new launched worker. choiceGeneratorAdvanced is a listener in which provided us the index of next choice JPF is going to process. Before we actually proceed to that state, we need to collect this information and update the path choice string.

- `State Listeners.` The most important listener related to state is stateBacktracked. This listener indicates after fully exploring a state, JVM has rolled back to the previous state and ready to check next choice of Choice Generator. For our approach, when such a listener is trigger, we update the path choice string by popping out the last choice of it. More importantly, for later processes, we need to stop the exploration once it is backtracked from the assigned frontier state to avoid exploring irrelevant paths. Besides this listener, we also used stateAdvanced to collect the depth of current state, in order to help controlling the flow of exploration.

IV. EVALUATION

In this chapter, we first discuss the tool support, our assumptions and artifacts for experiments, metrics, and research questions. Then we analyze our experimental results and have some further discussions of the evaluation.

Tool Support

We have implemented our technique in SPF. We have customized Memoise so that the trie nodes corresponding to assertions are treated as frontier nodes, and each worker leverages Memoise analysis to check the assertion along the target path(s) which are feasible and lead to the checked assertions. Symbolic execution is guided by the target path and constraint solving is turned on only as the frontier node is reached.

For environment, we have used LoneStar 5 on Texas Advanced Computing Center (TACC) cluster. The configuration of our computing node is:

- Dual Socket
- Xeon E5-2690 v3 (Haswell) : 12 cores per socket (24 cores/node), 2.6 GHz
- 64 GB DDR4-2133 (8 x 8GB dual rank x8 DIMMS)
- No local disk
- Hyperthreading Enabled - 48 threads (logical CPUs) per node
- JDK-1.8

We have chosen the latest version of JPF/SPF to support the JDK-1.8 environment. Also, we modified our implementation so that we can correctly distribute the jobs on different computing nodes. The communicating and allocating of jobs are controlled by TACC operation system and we have confirmed its effectiveness.

Assumptions

We made some assumptions when evaluating our approach. The following data to evaluate are collected based on these assumptions. We are aware of the risk that some potential threats to validity existed, and we discuss them in detail in later sections.

- A basic problem of symbolic execution is path explosion, which means if there are too many states and paths, traversing the whole state space trie could be very expensive. To solve this problem, SPF has implemented an optimization not to register a choice generator if there is only one or none choice is feasible. By doing this, infeasible states are ignored. Further more, if a state only had one child-state (when there is only one feasible choice for a choice generator), these two nodes are merged into one in state space trie. To fairly compare our technique with conventional symbolic execution and to avoid the possible confusion caused by this optimization, we have edited the code in SPF and disabled this optimization. Our artifacts are relatively small, and thus the introduced cost of traversing states is small. In addition, this change impacts on both conventional symbolic execution and our technique. Therefore, we consider this change acceptable. However, we also point out that this change is only to facilitate a good comparison between conventional symbolic execution and our technique, and does not impact the core algorithm of our approach.

- Another assumption we made is that we have enough resources to allocate for all workers to run at the earliest time possible. In this thesis, we do not discuss work-stealing between workers. In practice, we have noticed that such a strategy may have a negative impact on the performance of our approach. Such impact will be discussed later in this chapter.

```java
public class AssertionTest {
    // The method you need tests for
    public static int myMethod(int x, int y) {
        int result = 0;
        if (x < 0) {
            x = -x;
        }
        if (y < 0) {
            y = -y;
        }
        assert (x > 0); //assertion 1
        assert (y > 0); //assertion 2

        result = x + y;
        assert (x >=0 && y >=0 && result > 0);  //assertion 3
        return result;
    }
```

Figure IV.1: Source Code of AssertionTest

Artifacts

In our evaluation, we use one artifact with one manually created assertions and three other artifacts with assertions transformed from dynamically inferred invariants. These artifacts we chose are used in multiple research projects related to symbolic execution techniques.

The first artifact is a simple Java method named AssertionTest, which is shown in Figure IV.1. This method takes as input two integer values, and returns the sum of their absolute values. We manually created three assertions in different locations.

The second artifact is Wheel Brake System (WBS), which comes with official jpf-symbc package, and has been used in the literature for evaluating symbolic execution techniques. It is a synchronous reactive component from the automotive domain. This method determines how much braking pressure to apply based on the environment. It consists of one class and 231 lines of code. Since the original WBS code does not have assertions, we use Daikon (Ernst et al., 2007) to dynamically infer invariants for the program, and transform them into Java assert statements. As the eight invariants generated by Daikon, representing post-conditions, are all for the exit points of a method, we move the assertions synthesized from these invariants to different locations of the code

which are randomly selected. For this artifact, we do not care if the assertions are valid or not as long as the subject can be successfully compiled.

The third artifact is an open-source Java subject named `trityp`. This method takes three integer inputs which stand for the length of three edges of a triangle, and return an integer indicating the type of the triangle (scalene, isosceles, equilateral, or not a triangle). For this subject, ten assertions are manually inserted by the creator and put at the end of the source code. Unlike what we did to artifact WBS, we paid attention and made sure these assertions are not violated when me were moving the assertions to different branches of the method.

The last artifact is `Apollo` (RJC), which is also a Java subject used for evaluation in multiple former papers (Person et al., 2011). The Apollo Lunar Autopilot is a Simulink model that was automatically translated to Java. The translated Java code has 2.6 thousand lines of code in 54 classes. The Simulink model was created by an engineer working on the Apollo Lunar Module digital autopilot design team. The goal was to study how the model could have been designed in Simulink, if it had been available in 1961. The model is available from MathWorks6. It contains both Simulink blocks and Stateflow diagrams and makes use of complex Math functions (e.g. Math.sqrt). For this artifact, solving constraints is more expensive as they involve nonlinear calculation. We randomly pick up 7 assertions generated by Daikon, which are valid, invalid or irrelevant to the subject, and move them to different locations of the code.

Metrics and Research Questions

Our evaluation focuses on answering following research questions:

- RQ1: Correctness - Can our technique capture all violations detected by conventional symbolic execution?
- RQ2: Efficiency - Can our technique reduce the time cost comparing to normal symbolic execution?

In order to answer these research questions, we run conventional symbolic execution on our subjects and collect data for comparison. We measure our technique and conventional symbolic execution in following metrics:

1. Time cost: Total time used from start of exploration to exploration finished. We have insert instructions in listeners to output the system time when they are triggered, and calculate the result in milliseconds.

2. Memory cost: Total memory used by symbolic execution. We collected this metric from report of JPF engine. Calculated in Mega Bytes. Noted that since there is no efficient way to monitor the actual number of memory used by multiple workers, we just collect the cost of each worker.

3. Number of states explored: The number of states explored by each worker. We would like to assure each worker prunes irrelevant states before it reaches the assigned frontier state.

4. Number of solver calls: We implemented a counter to collect the number of times constraint solver is called by SPF.

5. Number of assertion violations detected: To ensure the correctness of our technique, we need to monitor the actual number of assertion violations detected. We must make sure that every violation reported by default symbolic execution is also detected by our approach.

Results and Analysis

The results shown in Table IV.1, Table IV.2, Table IV.3 and Table IV.4 are to evaluate the performance of each worker and the overall performance of our approach. For comparison, we have run conventional symbolic execution on the same artifacts, which explores the whole state space within given boundary. For the results, we first show the results of running conventional symbolic execution (SPF). Then, we show the data we gathered from each worker, starting

from the first worker we marked as Worker 0. This is the worker we launched in the first stage. Then, the results of each of worker in the second stage are shown in different lines. Finally, we calculate the overall performance of our approach at the end of the table. We gather and calculate the overall performance as following:

1. Time cost: The overall time cost of our approach is calculated by computing the time used from starting the first worker to all launched workers finished exploration.

2. Memory cost: Currently, we do not care about the overall memory cost of our approach, nor are we aiming to reduce or limit this cost. Besides the fact that parallel processing algorithms are commonly expensive in memory cost in order to speed up the computation, the memory usage reported by SPF may vary due to the underlying garbage collection.

3. Number of states explored: We checked the report of each worker, and added up the number of different states explored among them.

4. Number of solver calls: For this metric, we simply added up the numbers of constraint solver calls by each worker.

5. Number of assertion violations detected: We added up the numbers of violations detected by each worker.

The results of experiment on our first object, AssertionTest, are shown in Table IV.1. For this artifact, our technique detects 4 states corresponding to assertions by the first worker we dispatched. Thus, 4 more workers are launched one by one when such states are found.

As we can see from the table, each of our worker has explored only part of the state space trie as we designed. Also, with Memoise technique, newly launched workers did not call the constraint solver before reaching the frontier state it is assigned to. As a result, the time cost of each worker has decreased, and finally

Table IV.1: Results of AssertionTest

	TC (ms)	**MC (MB)**	**States Explored**	**Solver Calls**	**Violations Detected**
SPF	420	123	47	46	4
Worker 0	119	123	7	6	0
Worker 1	242	123	13	10	1
Worker 2	289	123	13	10	1
Worker 3	229	123	13	10	1
Worker 4	223	123	13	10	1
Overall	340	N/A	47	46	4

contributed to a reduction in time cost for the overall performance comparing to normal symbolic execution.

Each of these workers only explored part of the state space trie and called the PC solver only once (when the state standing for violation is reached). By reducing the number of state explored and solver called, the time cost has also been significantly reduced. Meantime, the memory cost of each worker remains the same, which is the minimum number we can get in our environment. Also, all 4 violations reported by conventional SPF listener are captured among workers.

For subject WBS, as shown in Table IV.2, 8 new workers are dispatched. For this subject, the differencing in time reduction is large between workers. The amount of reduction is affected by number of states explored, the complexity of each state and the complexity of constraint solving in each worker, but the total time cost of our approach is still less than conventional symbolic execution. Different with artifact AssertionTest, some of the workers detected more than one assertion violation, but the total number of violation detected is still 12, which is the same number reported by conventional symbolic execution. For this subject, we also noticed that some of the workers have used more memory than conventional symbolic execution regardless the fact they just explored part of the whole state space trie. We should mention that memory cost reported by JPF highly depends

Table IV.2: Results of WBS

	TC (ms)	**MC (MB)**	**States Explored**	**Solver Calls**	**Violations Detected**
SPF	612	966	127	126	12
Worker 0	303	966	53	52	0
Worker 1	86	966	4	2	1
Worker 2	198	966	16	8	1
Worker 3	217	1218	21	14	3
Worker 4	143	966	23	10	0
Worker 5	231	1218	33	22	1
Worker 6	261	966	46	4	1
Worker 7	172	966	27	12	4
Worker 8	243	966	15	2	1
Overall	477	N/A	127	126	12

Table IV.3: Results of trityp

	TC (ms)	**MC (MB)**	**States Explored**	**Solver Calls**	**Violations Detected**
SPF	158033	334	1085	1084	0
Workers	3182 to 128214	332 to 334	13 to 786	2 to 188	0
Overall	134852	N/A	1085	1084	0

on the timing of garbage collection of Java, and may vary among different runs.
Table IV.3 shows the result of running our technique on subject trityp. Since the number of assertion states is very large, which results in 97 frontier is found by the first worker, we cannot show the detailed performance of each worker. Instead, we show the range of each metrics among all workers. We have found that although we still achieved a reduction in time cost, the reduction rate is much smaller than former artifacts.
The result of testing is shown in IV.4. For this experiment, totally 88 new

Table IV.4: Results of Apollo

	TC (ms)	**MC (MB)**	**States Explored**	**Solver Calls**	**Violations Detected**
SPF	82825	123	867	866	21
Workers	86 to 27412	123	54 to 332	16 to 94	0 to 8
Overall	27859	N/A	867	866	21

workers has been launched and detected 21 violations. Considering the number of workers, we are still just showing the range of each evaluation metrics, as the same of artifact trityp. The reduction rate of time cost on this artifact is significantly larger than all other three artifacts. Among all the possible reason leading to this result, we found that constraint solving in this artifact is much more expensive than exploring the state space trie, for it needs conducting several nonlinear calculation.

From the result of these evaluations, we have proved that our technique achieved a full coverage of state space trie, while the number of violations detected being the same with when using conventional symbolic execution. Each of the worker only explored a part of state space trie, and Memoise helped to avoid redundant work of solving constraints of explored part of the trie. So, up to this point, we can answer the first research questions:

- RQ1: Correctness - Can our technique capture all violations detected by conventional symbolic execution? Yes. Our approach ensured the full coverage of state space trie and detected all violations found using conventional SPF listener.

- RQ2: Efficiency - Can our technique reduce the cost of symbolic execution in terms of time? Yes. Each worker in our technique used less time and reduced the number of states explored and PC solver called, and the

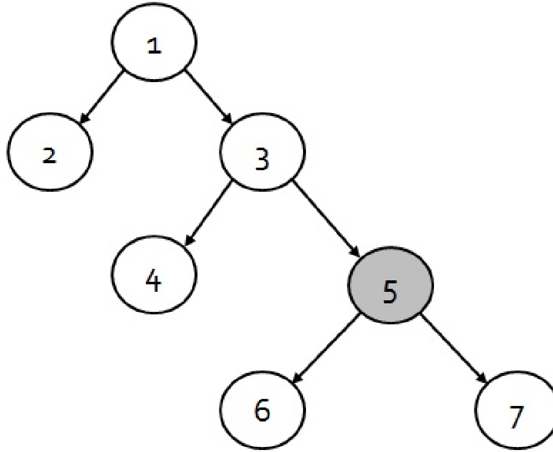

Figure IV.2: Example trie leading to worse performance

overall time cost has reduced.

Discussions

In addition to our evaluation, we would like to have a deeper discussion of some problems we noticed while implementing and evaluating our technique.

- `Visibility of Byte Code.` Since JPF is running on byte code, it is essential to conduct the analysis of SUT's byte code and mark out the instructions corresponding to assertions. In our experiments, we simply used javap -c to generate the byte code of subject. However, this method is not always feasible, for encapsulation mechanism of Java may hide the byte code of private methods, even if when users have full access to the source code. To get byte code with private methods, users need to change it to public and change it back later, which is very inconvenient.

- `Possible worse performance.` In our evaluation, we have seen that the performance may vary when using our approach on different artifacts. Upon researching the possible characteristics that may impact on efficiency, we found that in some cases our technique to get a worse performance than conventional symbolic execution.

  Figure IV.2 shows an example trie. State 5, the gray node, is a state

standing for assertion checking. When using conventional symbolic execution, it conducts depth first search of the trie, exploring state one by one in order of 1-2-3-4-5-6-7, and then respond to user with the result. With our approach, the first worker also conduct depth first search through 1-2-3-4-5. Then since state 5 is a frontier state, it starts a new worker and go through 1-3-5, and then state 6-7 to get the same report. In other words, if we run both conventional symbolic execution and our approach at the same time, we will find that while conventional symbolic execution is busy solving the path constraints leading to state 6, our technique started again from the root state and re-explored state 1-3-5 again. As a result, it is very possible that our new worker just start solving the path constraint while conventional symbolic execution has already reported to users. There are several possible characteristics that may impact the efficiency, i.e. complexity of state, cost of constraint solving, the structure of trie, and the location of assertion state. However, we still need to conduct more evaluation to see what is a suitable artifact to use our technique, and what is the decisive characteristic of it that may lead to a better or worse performance comparing to conventional symbolic execution.

- `Resources allocation.` We run our experiments in evaluation based on an assumption that we have enough resources to run all workers simultaneously. Which means, we assigned each of the worker a particular processor to explore the state space trie. It is feasible for our experiments, because the artifacts are relatively small for TACC to assign enough computing units as we requested. However, as the artifacts become bigger, it will be impossible to assign a different core to each worker. If multiple workers are forced to share a same set of computing resources, the efficiency of our technique will significantly reduce.

  Table IV.5 is an example experiment we run on subject WBS. Left column in the table is the time cost when all of them shared one processor, and

Table IV.5: Time cost when workers running on same/different cores.

	Sharing Time Cost (ms)	**Not Sharing Time Cost (ms)**
Worker 0	270	198
Worker 1	2209	181
Worker 2	1919	186
Worker 3	1974	178
Worker 4	1505	183
Worker 5	1602	217
Worker 6	2023	186
Worker 7	1769	206
Worker 8	781	179
Worker 9	1780	207
Worker 10	1913	193
Worker 11	1329	250
Worker 12	1400	188

right column is the time cost as they occupied their own processor. As we can see, when sharing the same processor, the same worker could use as more than 10 times of time as when it is using its own computing resources. The reason is that by not allocating new resources to each worker manually, all the scheduling of memory, processor and other resources are totally handled by operation system, and switching between workers are expensive and unpredictable. The more workers we launched at the same time, the higher the risk of a poor performance would happen. On the other hand, scheduling workers will increase the amount of communication between computing units, which could also be expensive. We have planned to implement a better algorithm to solve this problem and find the point of balance to reach the best performance possible.

- `Path Explosion.` As we mentioned in earlier part of this paper, we have disabled the optimization introduced by JPF in order to get a more plain result of conventional symbolic execution to compare with the performance

of our approach. However, we cannot deny that we also introduced a possible path explosion problem by doing this. The direct result is that the time cost in our evaluation could be more than using default symbolic execution implemented in SPF repository.

- `Limitation of symbolic execution.` Since our approach is based on symbolic execution technique, we cannot avoid the weakness and shortcomings of it. Our technique is aiming to improve the performance, trying to reduce impact of path explosion and expensive constraint solving by distributing exploration to different workers. Meanwhile, there are some problems we cannot solve by this technique. One of the shortcoming is symbolic execution cannot be used upon methods when the parameter cannot be "symbolized" i.e. strings, files, network communications, or data structures. Although research on this problem has been conducted and reached some progress (Fromherz et al., 2017), it is still far from reaching usability to be widely applied. This shortcoming limited the number of subjects which we can conduct symbolic execution on, and thus reduced the number of suitable target artifacts of our approach.

- `Alternative solutions.` In our current implementation, the newly launched processes are assigned with exploring all states beneath the frontier state. There is actually another solution, that after a process reached a frontier state, it also backtracks at the next frontier state it detected and dispatch another process. In other words, instead of conduction conventional symbolic execution from the frontier state, this process behaviors in the same way of the first process in our current approach. Due to the time limit, we did not implemented an executable version of this solution, nor did we conduct any evaluation comparing the performance. But in theory, this new solution could further reduce time cost when the state space beneath frontier state is deep, complicated, and more balanced in term of position of assertion checking states in the trie.

On the other hand, if the part of trie assigned with new process is shallow, or if the frontier states are too close to each other (i.e. when several assertions are put together in source code), our current solution might have a better performance than the alternate solution for the sacrifice of re-exploring unavoidable states would become too big as the work load between processes would become poorly using new solution. Another possible solution is the first worker does not stop exploring current path when a frontier state is detected. Instead, it sends the path choices towards the exact assertion violation state, which should be one of the children of frontier state, and choose to explore all other choices which may lead to other frontier states. The new launched workers just checks the feasibility towards the violation state and stop any further exploration of the state space. Comparing to our current solution, the workload of the first worker is bigger for now it is supposed to explore much more states. However, once an assertion checking is detected, each new worker needs less time before coming back to users, as the result of assertion checking can be reported to user as soon as it was checked. One possible problem of this alternate solution is that the first worker must figure out which choice of the choice generator is leading to assertion violation without exploring the state. Although in most cases it is the first choice (true branch of the if-instruction), we have already found some exceptions, especially when the assertion condition is complex.

V. RELATED WORK

Symbolic execution (King, 1976b; Clarke, 1976) performs systematic exploration of program behaviors for bug finding, and can be used to check the conformance of properties to program behaviors. Some recent projects (Yang et al., 2014; Zhang et al., 2014) have explored more efficient checking of assertions. Our work is complementary since it checks assertions in parallel and can potentially combine with these existing approaches to further improve the checking efficiency.

One of the task being wildly studied in this area is parallel symbolic execution. Several tools such as Cloud 9 (Bucur et al., 2011; Ciortea et al., 2010), Swarm (Holzmann et al., 2008; Groce et al., 2012) based on SPIN mode (Holzmann and Bosnacki, 2007), Simple Static Partitioning (SSP) (Staats and Păsăreanu, 2010) or ParSym (Siddiqui and Khurshid, 2010) are developed based on dynamic, static or hybrid algorithms. Although some of these techniques could archive a very considerable reduction compared with default non-parallel symbolic execution, they are not typically designed for checking properties. Different from these techniques, our technique is typically focusing on property checking, or assertion checking, using symbolic execution. While other frameworks balancing workload by simply using general boundaries, our technique use the location of assertion as "cut point" to distribute the trie in order to get earlier report about assertion violations.

On the other hand, assertions are very useful and are used to design several tools and used in checking system from different angles such as data structure (Berdine et al., 2006). Several tools aiming to generate likely assertions automatically have been developed such as Daikon (Ernst et al., 2007) as is used in our evaluation. Assertion checking is a suitable problem to be solved using parallel algorithms, since assertions are written to be side effect free which guarantees the checking result would remain the same. Thus, we are aiming to develop a distributed

symbolic execution technique to check assertions. While in our previous work, we introduced an approach to distributing the checking of assertions using static methods (Yang et al., 2015), this work is complementary and uses dynamic analysis to precisely locate the target paths reaching the checked assertions.

VI. FUTURE WORK

We plan to explore the following lines of work in the near future.

- `Static analysis.` Currently, overall our approach explores all states regardless of whether they are corresponding to property checking or not. Since our technique is aiming on checking properties only, exploring irrelevant states is unnecessary. Static analysis can be used to avoid the exploration of any paths or states that is not relevant to checking properties to further cut down the cost.

- `Reusing explored states.` By using symbolic execution, each worker could avoid exploring some of the states and go directly to its assigned frontier state. However, despite having been explored before, we could not avoid exploring all states on the path. This is decided by SPF, that we cannot start searching from the middle of the trie - every symbolic execution run in SPF has to start from the root state. In fact, since we did not change the code of SUT, re-exploring such states are redundant and not necessary. If we could find a way to avoid it i.e. implementing our technique on other frameworks, we can further reduce the cost and archive a even better efficiency.

- `Distributing with less resources.` Currently we assume the availability of enough computing resources that can run all processes at earliest time simultaneously. However, as the complexity of subject under analysis increases, a large number of target paths may need to be explored, which leaves us with the problem of finding an efficient way to distribute and schedule property checking when we do not have enough resources to run all the workers at the same time. Key algorithms from related researches (i.e. Cloud9) could be adapted to our technique and help solving this problem.

- `Analyzing bytecode automatically.`In our current implementation, the analysis of bytecode is done manually. As described in former parts of this paper, assertions are turned into a form of normal if-branch with exception after compiled. Althogh manually checking bytecodes can assure that we find all assertions without mistake, the efficiency is very low and not capable to be used in reality. To improve the usability of our approach, especially when the SUT is a complex project with multiple Java files, improvement on bytecode analysis is necessary. One way is to use third-party tools i.e. Apache-BCEL to help analyzing the source code and bytecode automatically at the same time, so that it will greatly reduce the manually effort needed before we start symbolic execution run.

- `Further evaluation.` Although we have used artifacts with different characteristics and proved the correctness and efficiency of our approach, we also noticed that the reduction of time cost differs dramatically between each of them. We would like to use more real world open-source Java programs to observe the performance of our technique, and try to find if there is a key characteristic of the artifact that could impact the effectiveness of our technique. Also, we can compare our current implementation with potential alternate solutions above, to further improve our technique to deal with different subjects.

VII. CONCLUSION

Symbolic execution is a powerful technique for checking properties. However, it could be very expensive due to problems like path explosion and time-consuming constraint solving. We focus on checking properties in the form of assertions in this work. To check all assertions in a program, users are usually forced to run symbolic execution sequentially for multiple times, or to wait for a long time before results of assertion checking are reported.

In this thesis, we used parallel analysis to reduce the time cost of checking assertions using symbolic execution. We implemented an approach on top of Symbolic Pathfinder to distribute the property checking workload among multiple workers and run them simultaneously. We conducted an evaluation with checking assertions in four different Java programs, and demonstrated the effectiveness and efficiency of our approach. In particular, our technique can achieve a better performance compared to conventional symbolic execution on assertion checking.

APPENDIX SECTION

APPENDIX A

Static Analysis of Properties

In our previous work (Yang et al., 2015), we have also developed an approach which uses static analysis to distribute properties to be checked using symbolic execution. Our key insight is that assertions should be side effect free as a rule, thereby deleting an assertion should not change the result of other assertions. Thus, we could go through the source code of SUT which has multiple assertions, and divide it into multiple sub-versions. Each of the sub-version keeps only one assertion activated.

After generating all sub-versions, we launch symbolic execution runs in parallel on each of them. The symbolic execution search is prioritized to explore shortest paths towards assertions so that earlier feedback on the checked assertions can be provided to users. A case study using two subject programs with manually annotated assertions shows that our approach for distributed assertion checking reduces the overall analysis time compared with regular non-distributed assertion checking using symbolic execution as implemented in SPF; and in sub-problem analysis which focuses on checking one single assertion, the guided and prioritized search can reduce explored states and constraint solving as well as can provide earlier assertion checking feedback.

One bottleneck of this technique is that going through all files of SUT could be very expensive. For example, we have to check every file in the repository regardless there are assertions in them or not. Another risk is that we have keep a large number of sub-versions, and the efficiency of copying a large file could be unavoidably bad.

Regardless the potential problems, this static analysis is still promising and could be further improved.

APPENDIX B

Listeners in JPF

Listeners are perhaps the most important extension mechanism of JPF. They provide a way to observe, interact with and extend JPF execution with our own classes. Since listeners are dynamically configured at run-time, they do not require any modification to the JPF core. Listeners are executed at the same level like JPF.

There are two basic listener interfaces, depending on corresponding event sources: SearchListeners and VMListeners. Since these interfaces are quite large, and listeners often need to implement both, JPF also provide "adapter" classes, i.e. implementors that contain all required method definitions with empty method bodies. Concrete listeners that extend these adapters therefore only have to override the notification methods they are interested in.

The adapter classes are used for the majority of listener implementations, especially since they also support two other interfaces/extension mechanisms that are often used in conjunction with Search/VMListeners: Property (to define program properties) and PublisherExtension (to produce output within the [[user:output JPF reporting system]]).

In our technique, we basically used VMListeners to control the order of exploration to skip, prune or backtrack from certain states. Meanwhile, we have noticed that the PublisherExtension cannot be triggered in our implementation. PublisherExtension is used to given a more detailed report about the method under test when the search is terminated. However, since our approach can also give the report of critical information (we can still get the counter-example when violation is triggered, even though not as easy to read as in PublisherExtension), we did not take effort to research on this small problem.

APPENDIX C

Daikon Invariant Detector

After searching real-world open source Java projects on several code-sharing repositories, we noticed except in shape of JUnit test suits, few open-source Java projects are using assertion features for some many reasons (e.g. extra effort it would be used to coding and difficulties in making assertions sound and side-effect free). Thus, we turned to use third-party software to generate assertions directly and move them around. We have chosen Daikon (Languages and Group) as the tool, which is used in several other papers on similar topics. Daikon is an implementation of dynamic detection of likely invariants; that is, the Daikon invariant detector reports likely program invariants. An invariant is a property that holds at a certain point or points in a program; these are often seen in assert statements, documentation, and formal specifications.
Daikon is freely available for download from the website. The distribution includes both source code and documentation, and its license permits unrestricted use. Many researchers and practitioners have used Daikon; those uses, and Daikon itself, are described in various publications.
After Daikon inserted the invariables into source code, they are usually put in a form of post-condition or pro-condition of certain methods. Thus, we need to move these assertions to the positions we want. In this paper, except some certain experiments, all assertions are put randomly in source code, and as long as the code can be correctly compiled, we do not pay attention to the correctness of them.

REFERENCES

Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., and Mcminn, P. (2013). An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001.

Berdine, J., Calcagno, C., and O'Hearn, P. W. (2006). *Smallfoot: Modular Automatic Assertion Checking with Separation Logic*, pages 115–137. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bucur, S., Ureche, V., Zamfir, C., and Candea, G. (2011). Parallel symbolic execution for automated real-world software testing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 183–198, New York, NY, USA. ACM.

Ciortea, L., Zamfir, C., Bucur, S., Chipounov, V., and Candea, G. (2010). Cloud9: A software testing service. *SIGOPS Oper. Syst. Rev.*, 43(4):5–10.

Clarke, L. A. (1976). A program testing system. In *Proceedings of the 1976 annual conference*, ACM '76, pages 488–491. ACM.

Clarke, L. A. and Rosenblum, D. S. (2006). A historical perspective on runtime assertion checking in software development. *SIGSOFT Software Engineering Notes*, 31(3).

Corbett, J. C., Dwyer, M. B., Hatcliff, J., and Robby (2000). Bandera: a source-level interface for model checking java programs. In *Proceedings of the 22th International Conference on Software Engineering*, pages 762–765.

Deng, X., Lee, J., and Robby (2006). Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 157–166.

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. (2007). The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45.

Fredkin, E. (1960). Trie memory. *Communications of the ACM*, 3:490–499.

Fromherz, A., Luckow, K. S., and Păsăreanu, C. S. (2017). Symbolic arrays in symbolic pathfinder. *SIGSOFT Softw. Eng. Notes*, 41(6):1–5.

Godefroid, P. (1997). Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 174–186.

Groce, A., Zhang, C., Eide, E., Chen, Y., and Regehr, J. (2012). Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 78–88, New York, NY, USA. ACM.

Havelund, K. and Pressburger, T. (2000). Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381.

Holzmann, G. J. and Bosnacki, D. (2007). Multi-core model checking with spin. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8.

Holzmann, G. J., Joshi, R., and Groce, A. (2008). *Tackling Large Verification Problems with the Swarm Tool*, pages 134–143. Springer Berlin Heidelberg, Berlin, Heidelberg.

Khurshid, S., PĂsĂreanu, C. S., and Visser, W. (2003). *Generalized Symbolic Execution for Model Checking and Testing*, pages 553–568. Springer Berlin Heidelberg, Berlin, Heidelberg.

King, J. C. (1976a). Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394.

King, J. C. (1976b). Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394.

Languages, U. P. and Group, S. E. The daikon invariant detector. `https://plse.cs.washington.edu/daikon/`.

Leavens, G. T., Cheon, Y., Clifton, C., Ruby, C., and Cok, D. R. (2005). How the design of jml accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55(1-3):185–208.

Meyer, B., Nerson, J.-M., and Matsuo, M. (1987). Eiffel: object-oriented design for software engineering. In *European Software Engineering Conference*, pages 221–229. Springer.

Ngo, M. N. and Tan, H. B. K. (2007). Detecting large number of infeasible paths through recognizing their patterns. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 215–224, New York, NY, USA. ACM.

Person, S., Yang, G., Rungta, N., and Khurshid, S. (2011). Directed incremental symbolic execution. *SIGPLAN Not.*, 46(6):504–515.

Păsăreanu, C. S. and Rungta, N. (2010). Symbolic pathfinder: Symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA. ACM.

Siddiqui, J. H. and Khurshid, S. (2010). ParSym: Parallel symbolic execution. In *Proceedings of the 2nd International Conference on Software Technology and Engineering*, pages V1–405 – V1–409.

Staats, M. and Păsăreanu, C. (2010). Parallel symbolic execution for structural test generation. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 183–194, New York, NY, USA. ACM.

Willard, D. E. (1984). New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, 28:379–394.

Yang, G., Do, Q. C. D., and Wen, J. (2015). Distributed assertion checking using symbolic execution. *SIGSOFT Softw. Eng. Notes*, 40(6):1–5.

Yang, G., Khurshid, S., Person, S., and Rungta, N. (2014). Property differencing for incremental checking. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1059–1070, New York, NY, USA. ACM.

Zhang, L., Yang, G., Rungta, N., Person, S., and Khurshid, S. (2014). Feedback-driven dynamic invariant discovery. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 362–372, New York, NY, USA. ACM.

Zhang, Y., Chen, Z., Wang, J., Dong, W., and Liu, Z. (2015). Regular property guided dynamic symbolic execution. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 643–653.