View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by The University of Nebraska, Omaha

University of Nebraska at Omaha
DigitalCommons@UNO

Student Work

5-2013

UNIT-LEVEL ISOLATION AND TESTING OF BUGGY CODE

Sanik Bajracharya
University of Nebraska at Omaha

Follow this and additional works at: https://digitalcommons.unomaha.edu/studentwork

 Part of the Computer Sciences Commons

Recommended Citation
Bajracharya, Sanik, "UNIT-LEVEL ISOLATION AND TESTING OF BUGGY CODE" (2013). *Student Work*. 2880.
https://digitalcommons.unomaha.edu/studentwork/2880

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.

UNIT-LEVEL ISOLATION AND TESTING OF BUGGY CODE

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfilment
of the Requirements for the Degree

Master of Science in Computer Science

University of Nebraska at Omaha

by

Sanik Bajracharya

Omaha, Nebraska

May, 2013

Supervisory Committee:

Harvey Siy, Ph.D.

Mahadevan Subramaniam, Ph.D.

Dhundy Bastola, Ph.D.

UMI Number: 1535891

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI 1535891

Published by ProQuest LLC (2013). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

UNIT-LEVEL ISOLATION AND TESTING OF BUGGY CODE

Sanik Bajracharya, M. S.

University of Nebraska, 2013

Advisor: Harvey Siy, Ph.D.

In real-world software development, maintenance plays a major role and developers spend 50-80% of their time in maintenance-related activities. During software maintenance, a significant amount of effort is spent on finding and fixing bugs. In some cases, the fix does not completely eliminate the buggy behavior; though it addresses the reported problem, it fails to account for conditions that could lead to similar failures. There could be many possible reasons: the conditions may have been overlooked or difficult to reproduce, e.g., when the components that invoke the code or the underlying components it interacts with can not put it in a state where latent errors appear. We posit that such latent errors can be discovered sooner if the buggy section can be tested more thoroughly in a separate environment, a strategy that is loosely analogous to the medical procedure of performing a biopsy where tissue is removed, examined and subjected to a battery of tests to determine the presence of a disease.

In this thesis, we propose a process in which the buggy code is extracted and isolated in a test framework. Test drivers and stubs are added to exercise the code and observe its interactions with its dependencies. We lay the groundwork for the creation of an automated tool for isolating code by studying its feasibility and investigating existing testing technologies that can facilitate the creation of such drivers and stubs. We investigate mocking frameworks, symbolic execution and model checking tools and test their capabilities by examining real bugs from the Apache Tomcat project.

We demonstrate the merits of performing unit-level symbolic execution and model checking to discover runtime exceptions and logical errors. The process is shown to have high coverage and able to uncover latent errors due to insufficient fixes.

ACKNOWLEDGMENTS

I would never have been able to finish my dissertation without the guidance of my committee members, and support from my family and wife.

I would like to express my deepest gratitude to my advisor, Prof Harvey Siy, for his excellent guidance, patience, and providing me with the excellent atmosphere for doing research. Without his guidance and peristent help this research would not have been possible.

I would like to thank Dr. Mahadevan Subramaniam for his valuable suggestions that helped me focus on important details for my research. I would also like to thank Dr. Dhundy Bastola for the encouragement and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the process of software development, software maintenance plays a significant role throughout the lifetime of the software product once it is released. According to IDC [18] software maintenance cost around $86 billion in 2005 alone, accounting for as much as two-thirds of the overall cost of software production. One of the main reasons for continuous maintenance activities is bug fixing. It is nearly impossible for complex software systems to be bug free. Though software systems can be tested with a large amount of test cases, it has been said that, "Program testing can be used to show the presence of bugs, but never to show their absence" [8]. Hence bug fixing is a constant task during software maintenance. It is estimated that developers spend 50-80% of their time fixing bugs. Efficient approaches to fixing bugs can free up the developers' time and reduce customer frustration. It will eventually also increase the software quality and minimize security loop holes.

1.1 Incomplete Fixes

To address this, we focus on the problem of incomplete fixes. During software maintenance, a significant amount of effort is spent on finding and fixing bugs. In some cases, the fix does not completely eliminate the buggy behavior; though it addresses the reported problem, it fails to account for conditions that could lead to similar failures.

Gu, et al. [13] provide an example of an incomplete fix:

Listing 1.1: Sample incomplete fix

```
1 // no idea what to do if it is a TAIL_CALL
2 if (fun instanceof NoSuchMethodShim
3     && op != Icode_TAIL_CALL ){
4
5     // get the shim and the actual method
6     NoSuchMethodShim =( NoSuchMethodShim ) fun ;
7     Callable noSuchMethodMethod =
8               noSuchMethodShim . noSuchMethodMethod ;
9     ...
10 }
```

JavaScript invokes `nosuchmethod` exception when an undefined method is called on an object. The above fix was done by adding an `if` block and this fix was the first fix for `nosuchmethod exception`. The fix takes care of `nosuchmethod` by invoking `NosuchMethodShim` and dispatches the undefined method on it. It passes the original test cases. However, the expression `op!=Icode_TAIL_CALL` could be false for an undefined method call. The developer missed this case and can prevent the program from invoking `nosuchmethodsim`. This fix eventually fails.

In practice, we often find such incomplete fixes when we search for reopened bugs. Reopened bugs refer to bug reports that have been closed but later reopened. To better understand reopened bugs, it is helpful to have a brief overview of the bug fixing process. One such process, followed by the widely used bug reporting system

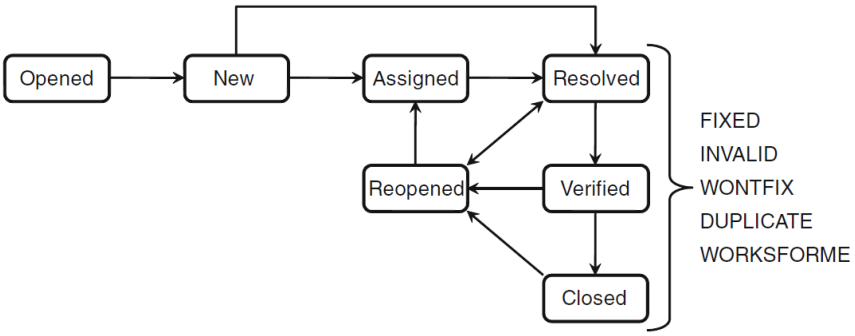


Figure 1.1: A common bug resolution process. Reprinted from [33].

Bugzilla, is depicted by the state machine in Figure 1.1. New bugs are assigned to a developer who resolves the bug. The resolution can be one of five options (`Fixed, Invalid, Wontfix, Duplicate, Worksforme`). The resolution is then verified by another developer/tester and if all involved are satisfied, it is closed. In many cases, the bug report might be reopened at some future time. In a survey of Microsoft developers, Zimmermann, et al. [39] gathered the following most common reasons for reopened bugs:

1. Bugs difficult to reproduce
2. Developers misunderstood root cause
3. Bug report had insufficient information
4. Priority of the bug has increased - "unimportant" bugs that were not fixed have now increased in importance
5. Regression bugs - bugs that were fixed earlier have reappeared
6. Process-related - caused by failures in following the process or miscommunication about the process

It should be noted that studying reopened bugs is not the same as studying incomplete fixes. According to Figure 1.1, only the first of the five possible resolution types is actually a bug fix. Furthermore, not all incomplete fixes lead to reopened bugs; some bug reporters will decide to open a new bug report instead. Nonetheless, the above list from Zimmermann, et al. [39] is instructive. In particular, it is easy to see how the first three reasons could lead to an inadequate fix. Additionally, the fifth reason usually indicates that the developer missed some corner cases. From these, we gather the possible reasons for incomplete fixes: the conditions may have been overlooked or difficult to reproduce, e.g., when the components that invoke the code or the underlying components it interacts with can not put it in a state where latent errors appear.

1.2 A Process for Software Biopsy

We posit that such latent errors can be discovered sooner if the buggy section can be tested more thoroughly in a separate environment, a strategy that is loosely analogous to the medical procedure of performing a biopsy where tissue is removed, examined and subjected to a battery of tests to determine the presence of a disease.

In this thesis, we propose a process in which the buggy code is extracted and isolated in a test framework. Test drivers and stubs are added to exercise the code and observe its interactions with its dependencies. We lay the groundwork for the creation of an automated tool for isolating code by studying its feasibility and investigating existing testing technologies that can facilitate the creation of such drivers and stubs. We investigate mocking frameworks, symbolic execution and model checking tools and test their capabilities by examining actual Java program bugs from the Apache Tomcat project.

Here is a sampling of the tools we investigated. JUnit is a popular and used unit testing framework which is used both for unit and integration testing. jMock, a mock library for java can is helpful and supports test driven development to fill the gap of dependencies that a program under test may have. We show that this approach of testing only executes some branch of the program leaving others untested and misses out important errors. In our work, we show the importance and advantage of using a method called symbolic model checking which executes the program under test dynamically with symbolic execution. The symbolic execution executes the program gathering constraints on inputs from conditional branches encountered along the execution path. The collected constraints are then systematically negated and solved with a constraint solver, whose solutions are mapped to new inputs that exercise different program execution paths. The tool we use for symbolic model checking is called Symbolic PathFinder [30], an extension of Java PathFinder [36]. In this tool, programs are executed on symbolic inputs representing multiple concrete inputs. The tool analyzes Java byte-code of the program under test, generates symbolic constraints satisfying each different branch. These constraints are solved using constraint solvers to generate test inputs guaranteed to achieve complex coverage criteria. We use this tool to test different programs and as a consequence list the advantage of using symbolic execution over other testing frameworks such as JUnit solely. We have also listed some of our recommendation and steps to be taken in order to efficiently use this tool and have high code coverage.

We demonstrate the merits of performing unit-level symbolic execution and model checking to discover runtime exceptions and logical errors. The process is shown to have high coverage and able to uncover latent errors due to insufficient fixes.

Other research have also highlighted the benefits of isolating tests [27] for uncovering latent errors in the code. Our work is similar, except we are not leveraging the

existing test suites but seek instead to generate new tests automatically.

Our work complements the approach proposed by Gu, et al. [13] to determine if a bug has really been fixed. Gu, et al. [13] go to great lengths to reduce the path explosion problem by narrowing down the test input space to traces within a certain edit distance away from the known buggy trace. We mitigate the path explosion problem by extracting only buggy code at a unit level (e.g., a method or a class). Furthermore, by stubbing the dependencies of the buggy code, it enabled us to control the outputs of these dependencies so as to test the behavior of the code after the bug fix, using conditions that may be difficult to reproduce in normal circumstances.

1.3 Organization

This thesis is divided into 5 chapters. Chapter 2 discusses on the existing research related to our thesis. It gives us idea how automated test input generation has been carried out so far and how can we use it real application. It suggests advantages of using the idea for testing unit based program and also highlight the limitation and ways that can be improved. Chapter 3 details our investigation into existing tools for test generation and mocking frameworks and explains how we can efficiently incorporate some of the tools. Chapter 4 explains the case studies conducted which supports our thesis. Finally, Chapter 5 discusses on the concluding part and what modification and addition can be used to effectively use model checking in software applications.

Chapter 2

Literature Review

In this chapter we will discuss different papers that provides background on bug localization, model checking, symbolic test input generation, mocking tools etc. A brief description of each of the tools will be presented following some examples that will show how they are used and what benefits that we can get from them for our research purpose.

2.1 Localization of Faulty Code

Bug localization is the process of finding the location of a reported bug. Typically, it is an effort-intensive process to locate a bug given only a description of the buggy behavior. Logically, identifying the bug location is the first step in isolating the buggy code fragment.

In our research, bug localization is done *post facto*; we examined bug reports that have already been resolved and manually went through the related code repository and search the file related to the bug code segment. This is only easy when the bug is located in one main module. Several scenarios complicate this approach, e.g., when

there are related files with dependent code element being used in the buggy code segment, when the code repository is huge, or the buggy code segment is long.

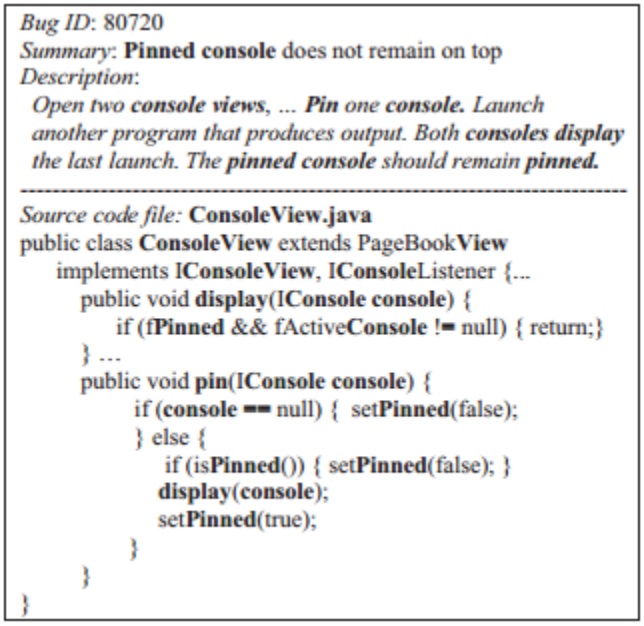
For these more generalized scenarios, other prior work in fault localization can be used. Several approaches to fault localization have been proposed in the literature: information retrieval, visualization, and analysis of execution traces. In this section we provide one example of each approach.

2.1.1 Information Retrieval Approach for Fault Localization

Zhou et al. [38] has proposed a tool called BugLocator, which is an information retrieval based method for locating the relevant files for fixing a bug. It ranks all the files based on textual similarity between source code and bug report using revised vector space model. A bug localization example is provided in Figure 2.1. This is taken from [38] and illustrates information retrieval based bug localization approach. The bug report in the example with `bug id 80720` has various parts such as description of the bug, summary etc. It is seen that the report contains many words such as pin, console, display, view etc. Hence, this bug appears related to console view. A source code file called ConsoleView.Java also contains similar words in it. The figure shows the match between the source code and the bug report literals. The words are matched with the source code repository and the source file with highest match is ranked first and so on. Developers can then search for related file one by one starting with highest rank.

2.1.2 Tarantula: Fault localization via Visualization

Jones et al.[20] developed a tool called Tarantula which is a fault localization tool that uses visualization techniques to improve developers ability to locate faults. Vi-



Bug ID: 80720
Summary: **Pinned console** does not remain on top
Description:
*Open two **console views**, ... **Pin** one **console.** Launch another program that produces output. Both **consoles display** the last launch. The **pinned console** should remain **pinned.***

Source code file: **ConsoleView.java**

```
public class ConsoleView extends PageBookView
    implements IConsoleView, IConsoleListener {...
      public void display(IConsole console) {
          if (fPinned && fActiveConsole != null) { return;}
      } ...
      public void pin(IConsole console) {
            if (console == null) {  setPinned(false);
            } else {
                if (isPinned()) { setPinned(false); }
              display(console);
              setPinned(true);
            }
      }
}
```

Figure 2.1: A bug report.

sualization is considered to be effective and promising for program exploration. The input to the visualization tool consists of three major components: the source code S; the pass/fail results of each t test-case in T for executing S; and the code coverage of the execution of S on each t in T. Example:

```
1    P    1 2 3 12 13 14 15 ...
2    P    1 2 23 24 25 26 27 ...
3    F    1 2 3 4 5 123 124 125 ...
```

The above shows a sample of the execution traces. On each of the lines, the first field is the test number t in T, second the pass or fail information of the test, and the trailing integers are the code coverage which is basically the source code line number executed by the test-case. The objective of this visualization tool is to assign different colors to the source code based on code covered by pass/fail test. It uses the line of pixels style code view introduced by the SeeSoft [9] system. The design main focus is to illustrate each line in the program executed by different sets of test case.

Color is used as a medium to represent which and how many different test cases have executed each line in the program source code. The selection of colors and applying it to provide a good visual mapping was a challenge to the developers of this tool. A line in the source code that is executed only by failed test cases are set to red color, the lines executed by only passed tests are rendered with green color, and the lines that are not executed by any test cases are left grey. However, there may be cases where a program line can be executed by both failed and passed test with equal number of occurrences, with different weights of occurrence by both failed and passed tests. Hence the researchers of this tool came up with a heuristic that represents each type of line by a different color (hue). The following equation is used to determine the color of the line for a statement s that is executed by at least one test case.

$$hue(s) = low\,hue(red) + \frac{\%passed(s)}{\%passed(s) + \%failed(s)} * hue\,range$$

$$bright(s) = max(\%passed(s), \%failed(s))$$

For example, for a test suite of 100 test cases, a statement s that is executed by 15 of 20 failed test cases and 40 of 80 passed test cases, and a hue range of 0 (red) to 100 (green), the hue and brightness are 40 and 75, respectively.

Tarantula tool is not available online at this time and it only handles C programs. It is being used internally by the researchers as to determine the effectiveness of their technique for locating faults in a program.

2.1.3 SOBER

Liu et al.[24] proposed a software fault checking model called SOBER that uses predicate ranking. The researchers use a statistical approach to localize software bugs without any prior knowledge of program semantic. It is different from statistical debugging approaches in that it only selects predicates correlated with program failures. SOBER uses predicate ranking by evaluating every patterns of predicated in both correct and incorrect executions respectively. It then suggests a predicate to be bug relevant if its evaluation pattern in incorrect runs is significantly different from the correct ones.

2.2 Code Isolation Using Mock Objects

A *mock object* is an object used in a situation where a stub is called for. Typically, mock objects are employed by developers when the code they are developing depends on some other object, code module or resource that is not yet available. In such cases, mock objects are created that comply with the interface of the real object and appear to behave in similar fashion. Mock frameworks facilitate the creation of mock objects and specification of some expected behavior. The mocking frameworks currently available generally differ in the way mock object behaviors are specified. For our research, mocking frameworks can be used to simulate the behavior of the objects the buggy code interacts with.

2.2.1 jMock

jMock is a library that supports test driven development of Java code with mock objects[19]. These mock objects help testers design and test the interactions between

the objects in the programs. We extract buggy code and isolate it. Then jMock is used to construct object referencing to different class that the code associates with. The use of mock gives us the flexibility to wrap the results as desired to the calling methods for testing. This way we are adding the possible expectations. An example is shown below on how mock is used.

Listing 2.1: Mock code skeleton

```
public void Connect(Connection dbCon) {
    //code here
}

//Mock
Connection dbConMock = new Mock(Connection.class);
Connect(dbConMock);
```

Suppose we have a buggy method called `Connect`. Eventually we will need to import `Connection` class to make use of it in the `Connect` method. The `Connection` is the environment dependency class. Inclusion of `Connection` class may require a need to pull the whole class to the isolated environment. This may lead to dependency and referencing problem i.e., if we pull the `Connection` class then this class may have reference to other class which will also need to pulled in too. Therefore, we use mock to delegate method call instead of original method. When using mock, an empty object obeying `Connection's` interface is created. In this class, dependent methods that are in use for the code execution are added. A mock object is created as a reference to this new `Connection` class. Expectations are added for each of the method used in `Connection` class to the mock object and desired results are analyzed and returned at runtime.

2.2.2 Moles

Moles is another mocking framework that provides test cases to run on isolation from environment dependencies[14]. Running test in isolation makes testing more robust, scalable, and reduces test execution time. Moles redirect calls to provided delegates instead of original methods. It provides features similar to mocking framework where dependencies are replaced with mock data references. Moles allow tester to substitute components at test time with codes that simulates its similar behavior which enables test case generation with Pex (discussed in Section 2.3.8).

2.3 Automatic Test Generation

Tool	Strategy	Main Engine
TestEra	bounded exhaustive generation	Alloy
Korat	bounded exhaustive generation	Korat
UDITA	bounded exhaustive generation	JPF
SPF	symbolic execution	Choco/CVC3
BLAST	symbolic execution	Simplify
Pex	symbolic execution	Z3
JCrasher	randomized tester	heuristics
SAGE	symbolic execution/fuzz tester	Z3

Table 2.1: Test generation tools studied.

Test generation seeks to automate the process of writing tests. Test writing requires much thought and effort especially to produce tests that give developers confidence in the reliability of their product. We are interested in test generation for similar reasons. In addition, test generation would enable us to automate the process of isolating buggy code by automatically generating the drivers that invoke the code to exhaustively reach all possible states in the code. In this section, we examine a cross-section of test generation tools. Table 2.1 lists the set of tools we studied.

This list includes many model checkers employing a variety of techniques for state exploration, including bounded exhaustive specification and symbolic execution. We introduce each of these tools and discuss their objectives and benefits.

2.3.1 Background: Software model checking

Model checking is the process of checking properties of hardware by exploring its state space. State space can be traversed much efficiently by considering large state for each single step. Having systematic state space exploration, it guarantees hardware/software to have exhaustive testing. In software, it is very effective for finding bugs as the model checker will search for all possible state space by analyzing different possible branch in the program. Thereafter, test inputs are generated for those state spaces. Model checking also accepts specification as input. Specifications are written in temporal logic. The specification are tested for its satisfiability and if it does not satisfy, a counterexample is generated against it.

There are research that have been applying model checking to software. Java PathFinder[36, 35] operates directly on Java programs and Verisoft[12] operates on C, C++ programs. Verisoft is a tool for systematically exploring the state spaces of systems. These state spaces are composed of concurrent processes that executes C program. Model checking is becoming popular for debugging and checking the correctness of a program. Complex systems have be modeled and then analyzed with the help of populates state space and verify them by applying it to the system for any failure that may occur.

There are other projects that uses software model checking; Bandera [5] and JCAT [7] translate Java programs into the input language of existing model checkers like SPIN [17] and SMV [25]. They handle a significant portion of Java, including dynamic

allocation, object references, exceptions, inheritance, and threads. They also provide automated support for reducing program's state space through program slicing and data abstraction. SLAM [3] uses predicate abstraction and model checking to analyze C programs for correct calls to API.

A key problem of model checking is dealing with state state explosion. For this, many complementary approaches have been proposed and widely used. Bounded exhaustive checking validates for all test inputs within a given bound [21, 26, 11]. Symbolic execution uses symbolic values instead of concrete ones [22, 30]. Modular model checking approaches also use the assume guarantee paradigm where the unit behavior is guaranteed as long as the environment outside the unit is assumed to follow a specified behavior [10, 31].

2.3.2 TestEra

Khurshid and Marinov[21] introduced a framework, TestEra, for automated testing of Java programs. It requires an input Java method and a pre post conditions of that method and a bound that specifies the size of the test inputs to be generated. TestEra uses pre condition to generate no isomorphic test inputs up to the specified bounds. The generated test inputs are used to execute the Java method for each test case. TestEra is based on Alloy tool which provides an automatic SAT-based tool form first order logic analysis.

TestEra is used to identify why a method failed the test. It uses the method's post condition to test any violation and reports with a counterexample. It also automatically generates the data structure from the method internal description and generates the data for it to work. For example, if a test method that performs a deletion in a tree, the input tree is automatically generated without a need to

manually construct the tree for test. This is very useful when it is hard to determine sequence of insertion in an empty tree if some deletion is needed.

Figure 2.2 an example that illustrates the main components of TestEra. A TestEra specification requires information such as method declaration, Java source file, class invariant, method precondition, method postcondition, and input bound size. These details are provided as command line argument to the TestEra main method to execute. As a result, three files are created and of these, two are alloy specifications. The first specification is for generating inputs and the other is for checking correctness. The third file is a Java test driver that translates Alloy instances to Java input objects, Java output objects to alloy instances, and executes the Java method for testing.

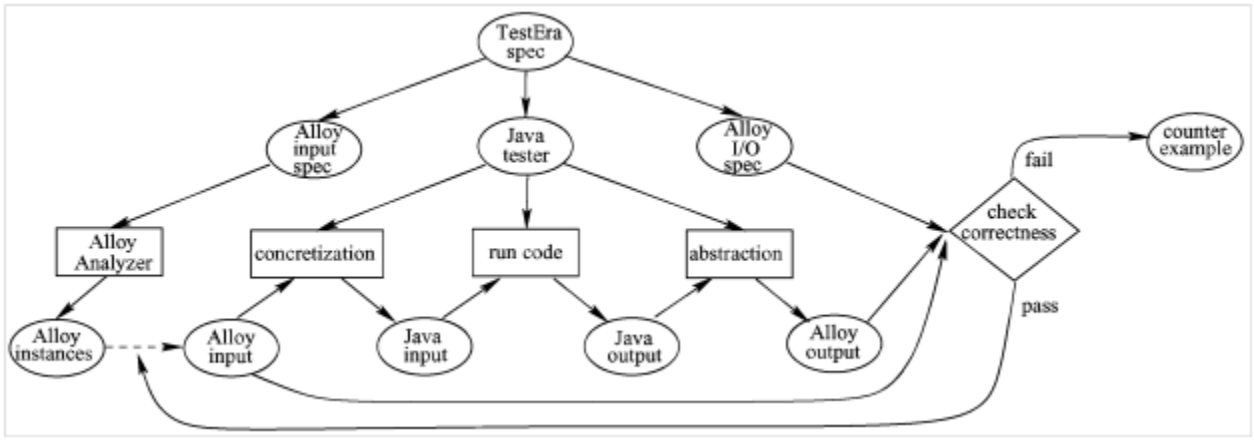


Figure 2.2: TestEra specifications.

2.3.2.1 How TestEra works

TestEra generates all non isomorphic test input as Alloy specifications for the use by the Alloy Analyzer. Each test input is tested against the Java method and then the method output is translated back to Alloy instance. The output Alloy instance and the original Alloy input instance are verified for their signatures and relations of Alloy

input/output. In case of verification failure, TestEra generates a counter-example for correct input value. If the verification succeeds TestEra continues further testing the method with next set of Alloy input instance.

2.3.3 Korat

Korat is an offshoot of TestEra, with tests and conditions written in Java rather than the separate Alloy language. Korat is a tool for constraint-based generation of structurally complex test inputs for Java programs. Structurally complex means that the inputs are structural and must satisfy complex constraints that relate parts of the structure[23] Korat requires an imperative predicate that specifies the desired structural constraints and a finitization that bounds the desired test input size. Korat generates all predicate inputs for which the predicate returns true. It is a way to filter the test input that are unnecessary. Korat performs systematic search of the predicates' input space and only generates the unfiltered inputs for exhaustive testing. A plugin called korattester was created that implements the Korat algorithm to generate test candidates for Java programs. The plugin comes with a user interface where bounds for inputs can be entered. This then with the help of imperative predicate (`repOk`), generates the test candidates. The test candidates are generated in an XML form. Korat then executes the method on each test case and uses the method postcondition as a test oracle to check the correctness of each output. Although korattester is enriched with these features, the ability to generate automated string inputs is still lacking. When compared to JUnit, there are disadvantages of JUnit testing framework. JUnit requires the tester to manually write every test cases which include Initialization of the object state that is to be tested Parameter specifications are required for the method that has it. The result should be verified with the

expected result value. JUnit unit test is performed only on a single state of the object through the use of setup method. The end result is what it cares about rather than the internal model logic. Overall this test can be categorized into a black box testing where test case passes on successful result match. Korat addresses these problems by generating non isomorphic test cases automatically up to a given bound. The advantage here is that, Korat can iterate the test method with the supplied bound, via finitization. These bounds are the all possible states of a test object withing a defined scope. Cons for Korat however is that given a complex test subject, generating test cases may take a longer period of time making it more difficult to frequently and repeatedly run tests [32].

2.3.3.1 How Korat works

After successful installation of korattester plugin, a menu called Korat is added as a plugin in eclipse. The plugin combines the benefits of the JUnit testing framework and the Korat algorithm. It allows the developers to use a wizard called finitizations, a scope of field ranges and Object pool sizes, for fields and objects within a selected Java class. This selection is under Korat menu and create state space sub menu. A user interface pops up where finitization is specified. At the end of the finitization wizard, the information collected is then used by Korat to generate all nonisomorphic states within the specified scope for that class which are stored to an XML file. The generated nonisomorphic test candidates passes through repOK method prior to being stored in XML. This is the place where the developer decide what kind of object states will be allowed for testing(e.g. acyclic or cyclic objects). The developer then writes JUnit test cases and uses a Korat Tester helper class which is used to run the test against all the generated test states. In each instance of test sates, it is able to make use of korattester pre and post object to make sure the tests pass. Any

false result will assume that test case to fail. Failure suggests that one of the test candidates failed the test. The output can be added with failure description in detail by pointing the cause of failure.

2.3.4 UDITA

Gligoric, et al. present UDITA[11], a Java based language with an interface for generating linked structures. It is used to generate test faster and make test descriptions easier to write than other frameworks/tools. UDITA is a delayed non-deterministic execution for model checking assembly code. Noll and Schlich[29] proposed the same concept but what makes UDITA different is the algorithm. Each execution of a test generation produces one test input. The execution engine explores all the possible execution for test input generations for bounded-exhaustive testing which validates the code that are under test. UDITA is not a fully automated system to generate test inputs because doing so will hinder tester to have sufficient control to define the space of desired test.

Overall UDITA is

- A new language for describing tests
- It implements a new test generation algorithms. Algorithm technique is built on systematic exploration performed by explicit state model checkers to obtain the effect of bounded-exhaustive testing. The efficiency is dependent upon lazy initialization of non-deterministic evaluation.
- It is implemented on top of Java PathFinder
- It is mostly used for black box testing.

2.3.5 Background: Symbolic execution

Symbolic execution uses symbolic values instead of using a concrete or actual value as the input for the program and hence the variable that take these values are represented as symbolic expressions. The output values computed by a program are expressed as a function of the input symbolic values [22]. The program state that is executed symbolically includes a symbolic program variable that contains symbolic data, a path condition (PC) and a program counter. The path condition is a boolean result statement over the symbolic inputs. It adds up the constraints for which inputs satisfy so that an execution can follow down a path. The program counter defines the next statement that is to be executed.

In Figure 2.3, the program on the left side swaps the value of the variable x and y if x is greater than y. On the right side of the figure, a symbolic execution tree is shown where PC counter is displayed. Initially, PC is true and variable x any y has symbolic values as X and Y respectively. PC gets updated at each branch according to inputs that is to be processed. After the execution of the first statement, there are two possible ways where program can proceed i.e., then and else of if statement. While taking these branches, the PC is updated accordingly. If the path condition becomes false, the symbolic state is not reachable and the program backtracks and the symbolic execution ends for the path. In the figure, statement (6) is unreachable.

2.3.6 Symbolic PathFinder

Khurshid, et al.[22] presents a novel framework for model checking and testing based on symbolic execution. Basically, there are two folds provided for symbolic executions to take place. First a source to source translation to instrument a program is performed using model checker. The instrumented program is then symbolically

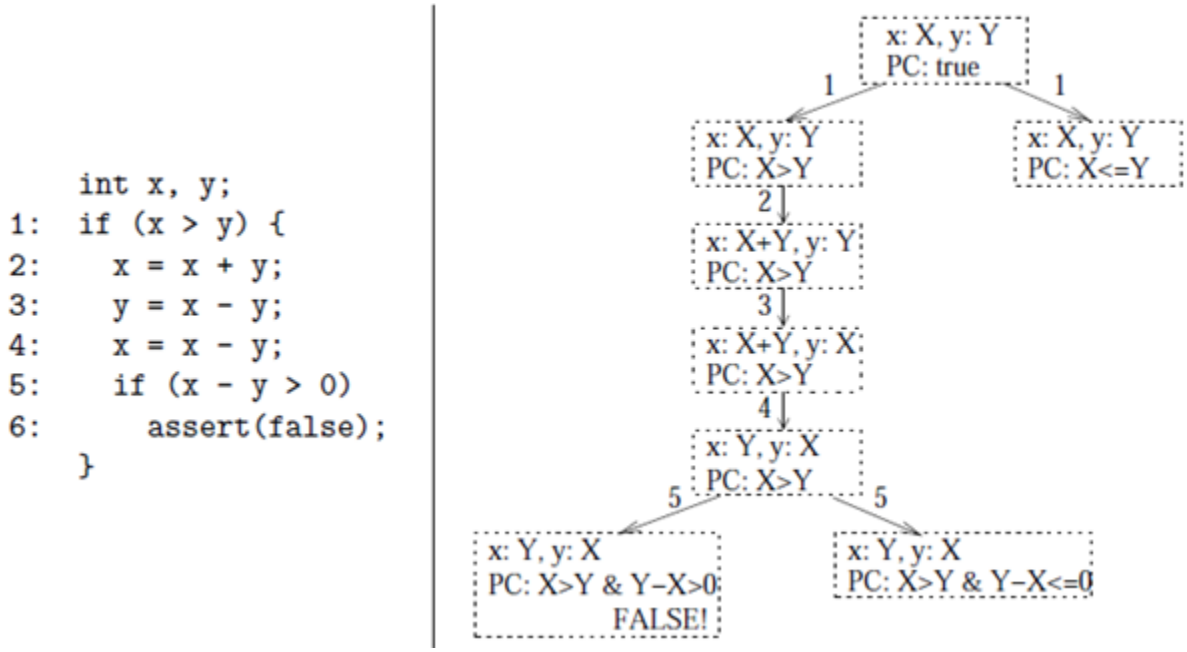


Figure 2.3: Left: Program that swaps two numbers. Right: Path condition of the program

executed by model checker. The program is checked automatically by model checkers to explore different paths and configurations using a decision procedure. Second, symbolic execution algorithm is used that handles dynamic data structures, method precondition, data, and concurrency. The algorithm uses lazy initialization to initialize components on the need basis without having to create a space bound beforehand. This work became the basis for Symbolic PathFinder.

Symbolic PathFinder combines symbolic execution with model checking and constraint solving for generating test inputs automatically [30]. It detects errors in Java programs for inputs that is not specified. It is one of the extended project of Java PathFinder. Program is executed in symbolic inputs that represents multiple concrete inputs. It analyzes Java bytecode, processes all possible path condition and comes up with a constraints for it that are satisfiable. Values for variables are represented as constraints generated. The constraints are solved by solvers to generate test inputs

guaranteed to achieve complex coverage criteria.

2.3.7 BLAST

Beyer, et al. [4] extended the software model checker BLAST [16] to automatically generate test suites. This generated test guarantees full coverage for the method under test with respect to a given predicate. Traditional model checking provides limited information and therefore does not ease where the programmer may wish to know the set of all program locations that can be reached for a given predicate. Researchers in this paper provide this information by making use of BLAST. Normally BLAST is used to find the reachable program locations and detect dead code. BLAST uses incremental model checking technology and its reuses counterexamples when possible. The extension of BLAST has been used to query C programs with 30K lines of code and was successful in finding dead codes, security and locking issues, and generating corresponding test suites.

Unit checking is close to the approach taken by these researchers where it generates a test vectors from traces. There are two phases where the first is Model Checking and later Test from counterexamples. Figure 2.4 is an example of a program which was used for testing against extended BLAST. The program takes three integers as input vectors and outputs the middle integer. The program does not have parenthesis and because of this the interpretation and evaluation by BLAST may not be correct. Proper parenthesis and nesting should be done before running extended BLAST[15] on it. To find a test vector that takes the program to L5, BLAST is used as the initial step to check whether the path is reachable to L5. BLAST uses iterative abstraction refinement procedure to check whether L5 is reachable and successively generates a counter example for it is reachable. Here L5 is reachable and the trace is given by

the following sequence of operations:

```
m = z; assume(y < z); assume(x < y);
```

In the second phase which is the tests from counter-examples, a counter-example trace from the model checking is used to find a test vector. First a trace formula which is the conjunction of constraints is built. In this example the formula is $(m = z) \wedge (y < z) \wedge (x < y)$. Second, the feasibility of the trace implies that the trace formula is satisfiable and we find a satisfying assignment to the formula. Therefore values such as $x = 0, y = 1, z = 2, m = 2$ gives a test vector that satisfies the constraint and takes the program to L5. These two phases are repeated for all the locations keeping in mind that one input can take several locations, until test vectors are set for all the location. The locations that fail to reach with possible test vectors are dead code or unreachable code. As an example the target location L12 can be reached with values (1, 0, 1) and BLAST can be a help to trace the location which in this case is <L1, L2, L3, L6, L10, L12>. In the control flow graph in Figure 2.5, the test vectors do not cover all the locations: L13 and L15. BLAST result infers that these locations are unreachable and hence there exists a dead code in the middle method. The reason for unreachable code is because of the missing braces pair in the source code and the misleading indentation. The if on L6 which was meant the if on L2 instead matches the else after L9.

2.3.8 Pex

Pex is an automated white-box test generation tool for .NET platform that uses dynamic symbolic execution to analyze code under test[34]. It validates unsafe memory access for individual execution path. It analyzes the program and determines the test input for parameterized unit testing and studies program behavior by monitoring the

```
#include <stdlib.h>
#include <stdio.h>

int readInt(void);

int middle(int x, int y, int z) {
L1:   int m = z;
L2:   if(y < z)
L3:     if(x < y)
L5:       m = y;
L6:     else if(x < z)
L9:       m = x;
      else
L10:   if(x > y)
L12:      m = y;
L13:   else if(x > z)
L15:      m = x;
L7:  return m;
}
int main() {
  int x, y, z;
  printf("Enter the 3 numbers: ");
  x = readInt();
  y = readInt();
  z = readInt();
  printf("Middle number: %d", middle(x,y,z));
}
```

Figure 2.4: A program to find a middle number

execution traces. Pex provides high coverage as it generates a set of test cases that overall satisfies every path through the program that is tested. Pex uses a constraint solver called Z3 to check execution path feasibility and creation of constraints. This tool is similar to Java PathFinder. The unique feature of Pex is:

- It can build solid symbolic representations of constraints which prove for execution path safety.
- It understands unsafe feature of .NET

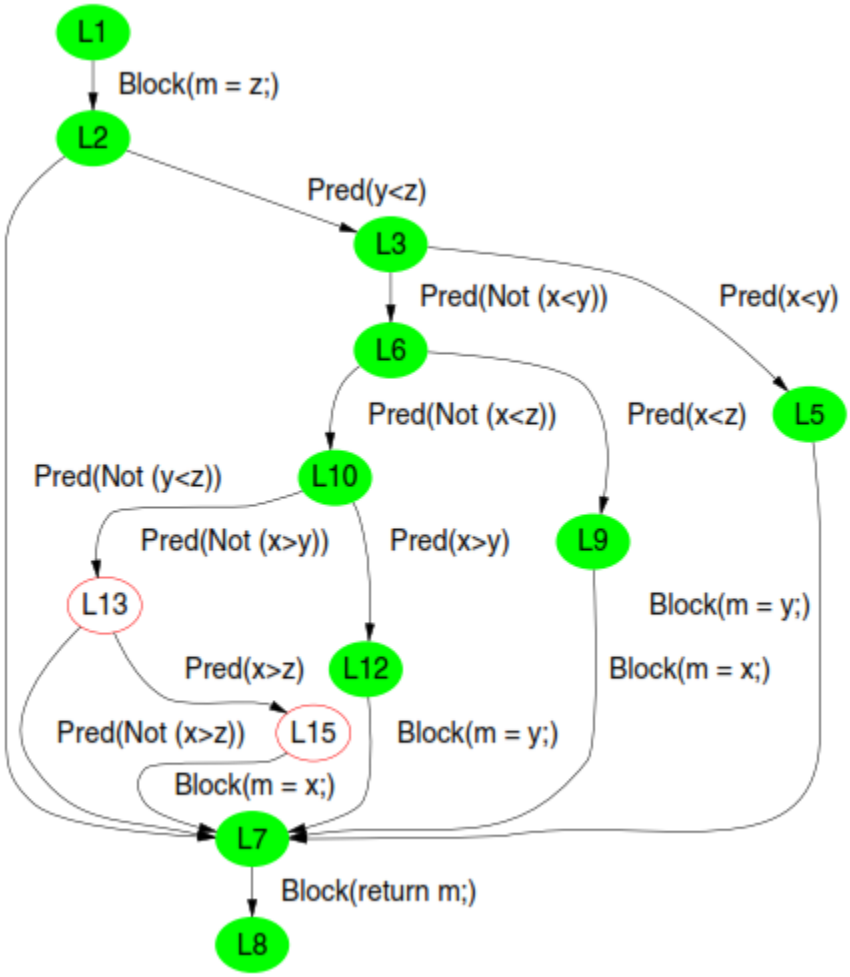


Figure 2.5: Flow graph of the program to find a middle number

- It applies its own search strategy to achieve high statement coverage

Pex uses a technique called dynamic symbolic execution. Dynamic symbolic execution extends conventional static symbolic execution. The additional feature of dynamic is that it collects information at run time which provides precise analysis.

2.3.9 Modular Model Checking: Assume guarantee model checking

Assume guarantee model checking is an approach to perform modular model checking of software units [31]. In addition to specifying the property of the software unit that to be validated, a specification of the behavior of its environment is also provided. The behavior of the software unit can then be verified assuming that the environment obeys its specified behavior.

Assume guarantee model checking[10] has been used to verify each thread in a multi-threaded program. Verification of each thread saves time and space while doing a model check. The test is targeted to loosely coupled software system. This new technique alleviates the need for model checking multi-threaded program with large state space. Assume guarantee takes care of environment assumption for every thread by providing a guarantee to each of those threads. The relation of this guarantee with those threads is initially empty and gradually gets filled up during the process of model checking. Assume guarantee keeps track of a global store and whenever any thread makes any changes to this global store, it adds the thread to a thread's guarantee. The thread gets added to the thread guarantee iteratively until the reachable state space. In other words, assume guarantee checks for any race condition, every reachable state satisfies the code invariant, and the assertion within the critical section does not fail for any thread. It means any variable such as static data that share data between multiple thread does not mix up resulting in incorrect data read and write. Below is an example where assume guarantee is used effectively.

Listing 2.2: Threaded program sample

```
int X = 1;
mutex m=0;
void p(){
    acquire;
```

```
    x = 0;
    x = x + 1
    assert x > 0;
    release;
}
```

All the thread accessing the method p() will first acquire the lock and manipulates variable x and release the lock. The variable x is protected by a mutex which is either the non zero identifier of the thread (thread being 1,..,n) holding the lock or else 0, if the lock is not held by any thread. Assume guarantee is incomplete and one aspect what it does not handle is the systems that create threads at run time.

2.3.10 jCrasher: Randomized Testing

Another tool called jCrasher[6] is an automatic testing tool for Java. It generates the JUnit testcases by examining methods used in a Java class and creates instances of different types to test the behavior of the method. The data generated for the parameters if any are random. As the name itself, it attempts to detect bugs by causing the program under test to crash. Some of the successors of jCrasher are Check 'n' Crash and DSD Crasher. Limitations of jCrasher are that the parameters in a method play major role. Without it jCrasher does know how to populate objects for ones that is used in a method. Numeric auto generations are simply positive, negative ones and zeros. jCrasher cannot populate string literals for testing. Most of the testing is either black box or regression testing. Both of these are effective techniques when there is a need to find out the inputs that is going to break the program. A black box testing is a test where a tester tries to interact with the application program without any idea of its internal workings. The tester just checks whether the input to the program gives a pre-known output without any errors. Whereas regression testing is a set of test suite that tests different parts of overall

functionality of the application. For each new version of the code, the set of test suite is ran against the regression test to see if it has any bug. jCrasher is a random testing tool for Java classes. It generates random but type-correct inputs in an attempt to cause a Java application to crash, that is, to throw an unexpected exception that is not consistent with good Java programming practices for signaling illegal inputs [6]. Use of random testing however has some advantages. Random testing does not require user test input and thus it is cheap. It can easily figure out boundaries to null pointer exception, array index out of bound, divide by zero exception etc. with this on hand, jCrasher offers some features that are unique when compared to other tools that are being researched for similar purpose: jCrasher constructs test cases at random . It takes Java type system to construct the random input data. jCrasher has defined a heuristic that knows whether the program bug has occurred due to jCrasher supplied inputs or by Java exception. jCrasher ensures that every test runs on a clean slate. This means if a static data in present in a program then change to it by previous test does not affect the current test. jCrasher produces a JUnit test file as an output. A tester finds that a test is good the he can integrate it in regression testing suite.

2.3.10.1 Working of jCrasher

For a given method M of a class C, jCrasher generates random test input sample according to the the parameters in the method. Each test case is created in a different way such that the parameter combination to M is considered. At first place, jCrasher uses Java reflection to identify method parameter types, the return type by method M, subtyping relations and visibility constraints. Later jCrasher determines what test cases should be generated and how many for each method M in class C. These test cases are output in a JUnit test class. Let's suppose we have a method M(A, int) with

parameters A as an object reference and an integer. The integer is either -1, 0, or 1 where as object reference is null or reference to that object. The test case generated is the total combination of these parameter mappings. Therefore some possible test cases with input test parameters are: M(null, -1), M(null, 0), M(null, -1), M(new A(), -1), M(new A(), 0).

2.3.11 SAGE: Fuzz Testing

Microsoft has taken great caution in finding out security vulnerabilities and releasing patches to fix them. Many of these vulnerabilities are the result of programming errors in code for parsing files and data transferred over the internet. Black box fuzzing, a simple and effective technique, is used to find security vulnerabilities in software. Many bugs are traced with this method and therefore Microsoft enforces this method to be used as one their bug finding step for every product. However, use of blackbox fuzzing has limitations because it has low code coverage and can miss security bugs.

Listing 2.3: Program illustrating limitation of blackbox fuzzing

```
int foo(int x) {
    int y = x + 3;
    if (y == 13) abort();
    return 0;
}
```

The then branch in the above conditional statement has only 1 in 232 chances of meeting it if the input variable `x` has a randomly chosen 32 bit value.

An alternative to blackbox fuzzing is a whitebox fuzzing. White box fuzzing is built upon systematic dynamic test generation and goes beyond unit testing. It uses symbolic execution that executes the program under test dynamically. The constraints on inputs are gathered for each of the conditional branch encountered

along the path of execution. The constraints are ruled out and are solved using a constraint solver that will provide solutions to produce new inputs that exercise different program execution paths. In the example above, the symbolic execution with an initial value of 0 for variable x will take the else branch of the conditional statement and thus it produces a constraint `x+3 <>13`. This constraint is then negated and solved and produces a value for x=10 that will cause the program to follow the then branch. This approach helps in solving security bugs even without knowledge of the input format. These may help in discovering errors related to buffer, memory allocation etc. The systematic dynamic test generation by use of symbolic execution is very helpful in achieving full path coverage, thus satisfying program verification. However, in practical, the search is incomplete because the number of execution paths in the program under test are generally huge and also because the symbolic execution, constraint generation and constraint solving can be imprecise due to complex program statements.

SAGE, Scalable Automated Guided Execution is being used at Microsoft for white-box fuzzing. It overcomes the issue of generating small number of test inputs for complex programs. SAGE implements a novel directed search algorithm that maximizes the number of new input test generated from each symbolic execution. As a result a single symbolic execution is able to generate thousands of new tests. SAGE was the first tool to perform dynamic symbolic execution. It is being used extensively at Microsoft and extends program analysis, testing, verification and model checking. It has discovered thousands of bugs related to security issues in many large applications used at Microsoft. This has saved them millions of dollars, time and energy by avoiding expensive security patches to more than one billion PCs.

We reviewed different papers that provide background that started with bug localization and entered into software model checking and symbolic test input genera-

tions. Tools that are helpful in mocking were also presented. These papers helped us in understanding each of their objectives and usefulness. While each of the methods described here are very interesting and innovative, we have chosen some tools for further investigation: jMock, Korat, and Symbolic PathFinder as it suits our research the best. In the next chapter, we describe these tools in more detail with examples and list out their advantages and limitations.

Chapter 3

Tool Investigations

In this chapter, we will discuss jMock, Korat, and Symbolic PathFinder in detail with some useful examples. We will focus on how they are used and what can they contribute to our research.

3.1 High Level Approach

Fixing bug and testing is a time consuming task and sometimes some fix does not seem correct. Researchers have been trying to study on automated test generation techniques and have created several tools to make bug fixing easier and efficient. However, there are certain limitations. We studied different methods that have been applied by several researchers. We also tested some tools and techniques applied by them. Hence, our approach is to utilize these tools in an effective way so that bug fixing process becomes easy and ensure code coverage with test inputs as well.

Bugzilla provides a repository for bug reports and these reports carry detailed information regarding a bug and discussions leading to their resolution. Figure 3.1 shows the high level approach that locates, isolates, and fixes a bug. Bug report

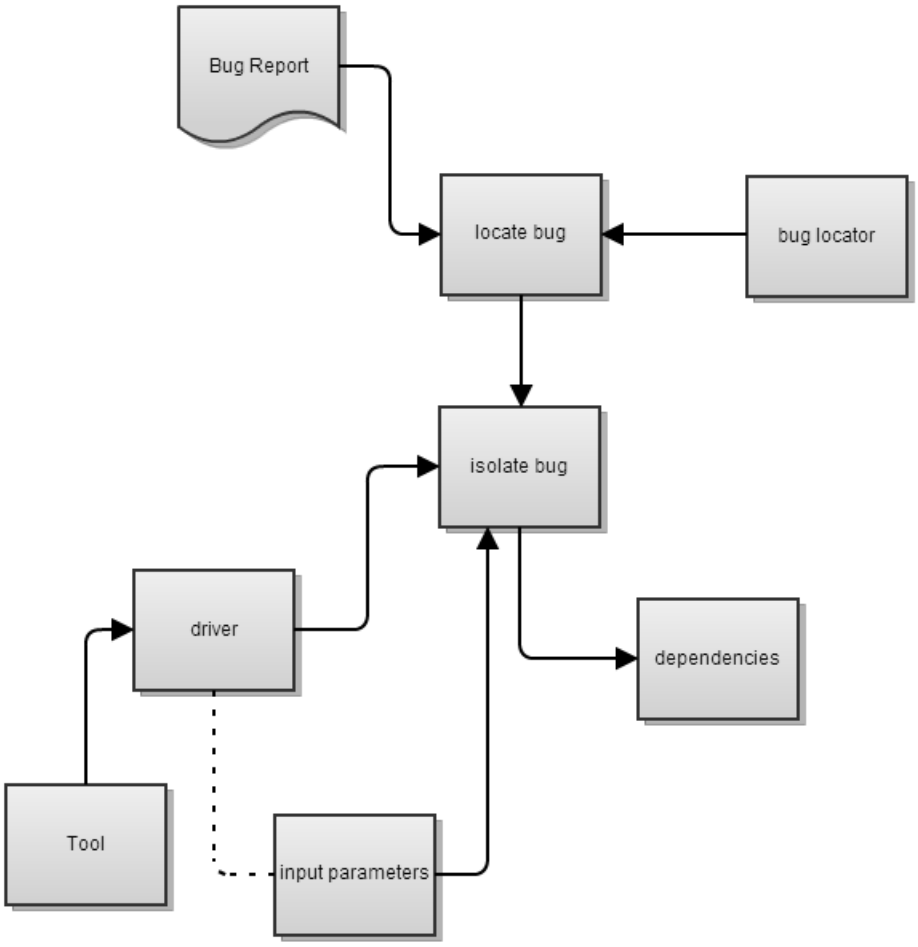


Figure 3.1: High level overview approach

provides information about the cause of the error and the files affected. A bug locator tool, such as Tarantula can be used to extract the bug from the source code. However, for the our study, we manually locate the file and isolate the buggy code segment. We then supply the necessary dependencies to the isolated code. Before isolating the code, we examine and perform a quick code flow check of the method that has bug. The code flow provides us with useful information on how much code should be isolated so that it guarantees bug reproduction in isolation. We use some tools to drive the isolated code segment. We will be using couple of tools for the test and

choose the best one that meet our goals. Necessary inputs to the code segment are provided during the process of testing.

The dependencies are applied by either creating stub classes or interfaces that the object within the buggy code segment refers to. The tools that we used so far are jMock, Korat, and Symbolic PathFinder. It depends upon the tool on how dependency will be provided. All these three tools have their own way of tackling bug and its own importance. However, Symbolic PathFinder has power for model checking and generated test inputs by symbolically executing the constraint in the object level.

3.2 JUnit

JUnit is the unit testing framework for Java programming language and it is commonly used in Java. We use this with some of the tools such as jMock and Korattester. JUnit acts as a driver for these tools and leverage code under test. We came across a memory leak bug and after studying the bug report. We wanted to test how JUnit alone can take care of the leaks.

Java has two classes: `ReferenceQueue` and `WeakReference`. When we create an object of a class eg. `(eg: Object ref = new Object())`, this is referred as a strong referenced (i.e. it is not eligible for garbage collector). At times, we do not release the memory occupied by "ref" (unused) and hence we suffer from memory leaks. The solution would be to declare "ref" as weak reference. Doing this we can force it to GC when we assign "ref" to NULL and later call System.GC. There are four types of reference: strong, soft, weak, and phantom. Unused reference are added to ReferenceQueue in order to transfer references to GC. Therefore in JUnit, we assign the object "ref" to weak reference, set it to null and garbage collect it. Then check

to see if the object has been enqueued for GC.

3.3 jMock

We discuss two different methods to locate and fix a bug. The first approach is to use mocking via jMock and the second is by using symbolic model checking. The purpose of this is to show how efficient and effective is symbolic model checking when compared to using jMock. However, mocking framework also has its own significance when it comes to stub, faking server data, etc.

3.3.1 Example

In Listing 3.1, the method `add` has a parameter called data which is a reference of `SendfileData` class. This object reference invokes methods of `SendfileData` class here. jMock is used to mock the object reference data and send desired value to it when it invokes its method (see Listing 3.2. To mock the object, `Mockery` class is used. `Mockery` class is then used to mock the desired class to create a mock object. This `Mockery` class instance adds *expectation* to the mocked `data` object. The expectations are the values that are set up in order to return for each call on method of mocked object `data`. These values are set up by the developers as required for testing. Inside expectations, a pair of `will` following `allowing` is an expectation. The `allowing` will add a method to a mock object and that means the mock object now can call that method. If the method has a return type, how do we return that? The `will` method accepts a parameter that is used to return values to the method called by the mock object. It can also have multiple return values one after another in a consecutive manner.

Although, objects can be mocked and provide mocked data when a service requests it, it has some limitations. It is inefficient as a fact that it does not provide a means to generating different inputs for testing.

3.3.2 Limitations

- The return values added to expectation are limited and needs to be added every time on a consecutive fashion
- The return values are manually picked.
- The code coverage by this method is low.
- Prone to error if the return list required is very long.

Listing 3.1: jMock example: method under test

```
public boolean add(SendfileData data) {
    data.setfdpool(this.s.pool(data.getsocket()));
    while (true) {
        long nw = this.s.sendfilen(data.getsocket(), data.getfd(),
        data.getpos(),      data.getend() - data.getpos(), 0);
        if (nw < 0) {
            if (!(-nw == EAGAIN)){
                this.pool.destroy(data.getfdpool());
                // No need to close socket, this will be done by
                // calling code since data.socket == 0
                data.setsocket(0);
                return false;
            } else {
                    // Break the loop and add the socket to poller.
                break;
            }
        }
        if (data.getpos() >= data.getend()) {
            this.pool.destroy(data.getfdpool());
            return true;
        }
        return false;
        }
}
```

Listing 3.2: jMock example: Mock objects and Expectations

```
Mockery context = new Mockery();
final File mockFile = context.mock(File.class);
final Pool mockPool = context.mock(Pool.class);
final SendfileData mockSendfileData = context.mock(SendfileData.class);
final Socket mockSocket = context.mock(Socket.class);

context.checking(new Expectations(){{
    allowing(mockSendfileData).setfdpool(with(any(long.class)));
    allowing(mockSendfileData).setsocket(with(any(int.class)));
    allowing(mockSendfileData).setfdpool(with(any(long.class)));
    allowing(mockSendfileData).setpos(with(any(long.class)));
    allowing(mockSendfileData).getsocket();
    will(returnValue((long)1));
        allowing(mockSendfileData).getfdpool();
    will(returnValue((long)1));
        allowing(mockSendfileData).getfd();
    will(returnValue(mockFile));
        allowing(mockSendfileData).getend();
    will(onConsecutiveCalls(
                returnValue((long)1),
                returnValue((long)1)));

    allowing(mockSendfileData).getpos();
    will(onConsecutiveCalls(
        returnValue((long)0),
        returnValue((long)0),
        returnValue((long)0),
        returnValue((long)1))); /*bcoz pos = pos + nw*/
}});

context.checking(new Expectations(){{
        allowing(mockSocket).sendfilen((long)1, mockFile, (long)0, (long
            )1, 0);
        will(returnValue((long)1));
        allowing(mockSocket).pool(with(any(long.class)));
        will(returnValue((long)1));
}});
```

3.4 Korattester

Korat is a tool that is used to execute the code segment with automated test input generation. Korat, unlike Symbolic PathFinder, accepts the list of state space boundaries to be assigned for each of the variables used in the program. At the end of

this step, Korat, with the help of the imperative predicate that specifies the desired structural constraints, generates all predicate inputs (within the bounds) for which the predicate returns true. The inputs are stored in an XML and during the execution of the program, this XML is used. The limitation of Korat is that it has no string manipulation feature and the test input are decided by the tester instead.

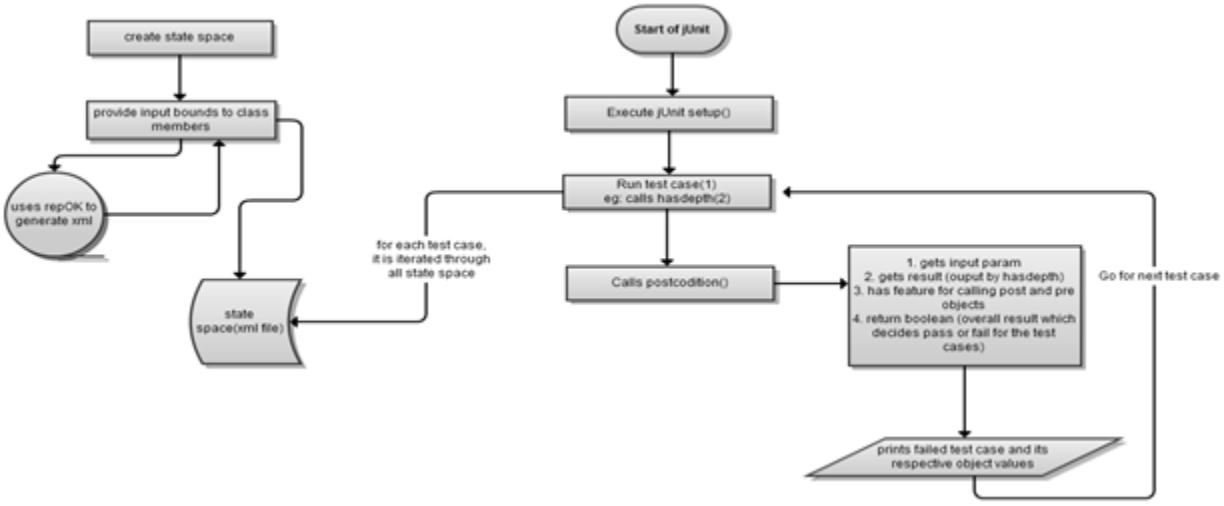


Figure 3.2: Working of KoratTester

With the help of Korattester, we run the test input generated by it in the form of XML. Korattester is run on top of JUnit and it then executes the method on each test cases and uses the method postcondition as a test oracle to check the correctness of each output. Figure 3.2 shows the testing process using Korattester.

Steps in Korattester:

- Create state space (isomorphic test inputs)
- Use repOk (predicate) to generate isomorphic test inputs and stores it in XML file
- Run each input on testing method. Test inputs are taken from XML file

- Use postcondition of Korattester to check for correctness
- Any failed test is printed out as a warning.

The results of our experimentation with Korattester shows the following limitations of Korat:

- It considers all the field properties of the class under test. There is no way to ignore a property from getting it picked up for test generation.
- String operation does not work as well as expected.
- It requires a imperative predicate which is a constraint to be implemented to filter unwanted test inputs

3.5 SPF: Symbolic PathFinder

In Listing 3.3, we isolate the method that has a bug in it. This is taken from Apache Tomcat 6. This code belongs to a single file commit. Hence, we do not have to change other file after making any fix to this buggy file. After isolation of the method, we implemented symbolic execution to fix the bug and to identify the possible loop holes.

Listing 3.3: SPF example: method under test

```
protected Connection open() throws SQLException {
    // Do nothing if there is a database connection already open
    if (dbConnection != null)
        return (dbConnection);

    // Instantiate our database driver if necessary
    if (driver == null) {
        try {
            Class clazz = Class.forName(driverName);
            driver = (Driver) clazz.newInstance();
        } catch (Throwable e) {
            throw new SQLException(e.getMessage());
        }
```

```
    }

    // Open a new connection
    Properties props = new Properties();
    if (connectionName != null)
        props.put("user", connectionName);
    if (connectionPassword != null)
        props.put("password", connectionPassword);
    dbConnection = driver.connect(connectionURL, props);
    dbConnection.setAutoCommit(false);
    return (dbConnection);
}
```

In the example above, the problem in the code snippet is a null pointer exception. The code that is highlighted is the place where the exception occurred. The reason for this exception is that when `driver.connect(connectionURL, props)` is invoked, the resultant connection was `null` due to an incorrect `connectionURL` for driver connection. Therefore, invocation of a method with a null object caused the error.

This shows that the bug is preserved even after the isolation. This provides us with an advantage of isolating any code segment that has a bug and use external tool to analyze the cause of the bug and measures to fix it.

We applied symbolic model checking to this isolated code segment. The way we started it is by first adding the dependencies that the method under test requires. We created Properties, Driver, and Connection interfaces so that the dependency is complete. Then we use Symbolic PathFinder's reference to add desired object to it. Adding these objects to symbc will generate possible test input for the object to run the program. Each iteration has different auto generated test inputs. In addition, any conditional branches used within the code segment are analyzed and respective test input matching the condition is generated. Listing 3.4 is the code segment after adding symbolic reference to it.

Listing 3.4: SPF example: code with symbolic references

```
public class JDBCRealm {
```

```
Connection dbConnection;
Driver driver;
int digit;
Util util;
//@Symbolic("true")
boolean hasURL;
String connectionURL = "http://dummysite.com/driver/";
String driverName = "com.dummysite.mysql.Driver";
String driverName;
String[] driverNameList;

public Connection open() throws Exception {
    dbConnection = (Connection) Debug.makeSymbolicRef("input_C",
        new Connection());
    driver = (Driver) Debug.makeSymbolicRef("input_D",
        new Driver());
    if(this.dbConnection != null){
        this.dbConnection.util = (Util) Debug.makeSymbolicRef("input_U",
            new Util());
        his.dbConnection.conIndex = Verify.getInt(2,3);
        this.dbConnection.status = Verify.getBoolean();
    }

    driverName =  this.driverNameList[this.digit];
    String connectionPassword = "testpass";
    String connectionName = "testuser" ;

    if (driver == null) {
        try {
            Class clazz = Class.forName(driverName);
            driver = (Driver) clazz.newInstance();
        } catch (Throwable e) {
            throw new Exception("error1");
        }
    }
    // Open a new connection
    Properties props = new Properties();
    if (connectionName != null)
        props.put("user", connectionName);
    if (connectionPassword != null)
        props.put("password", connectionPassword);
    //assert connectionURL
    dbConnection = driver.connect(connectionURL, props);
    dbConnection.setAutoCommit(false);
    return (dbConnection);
}

public static void main(String[] args){
    JDBCRealm obj = new JDBCRealm();
    obj.driverNameList = new String[]{"com.dummysite.mysql.Driver"};
    obj.hasURL = Verify.getBoolean();
```

```
        obj.digit = Verify.getInt(0,0);
        try {
            obj.open();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Symbolic execution uses symbolic values to execute the program instead of the actual (concrete) values. It is combined with model checking and constraint solving for the test case generation. Symbolic PathFinder uses JPF model checking tool (jpf-core) analysis engine. The analysis engine analyses numeric constraints from the generated code structure of the program. These constraints are solved and appropriate test inputs are generated to reach that part of the code. All these generated test inputs are introduced to the program to guarantee that it reaches every part of the program[22, 28].

In the above code segment, the codes that are highlighted are added after successful execution of the program using symbolic execution. For symbolic execution setup, we use class Debug of symbc. The makeSymbolicRef of the Debug class provides an advantage of setting up a variable to make it symbolic. In addition, these variable or objects added to symbolic execution are lazy initialized. It means the fields get initialized when they are first accessed during the method's symbolic execution. On verification of symbolic execution on the program, the program runs through symbolic values and provides each test output. This output if unsuccessful, returns an error. In this program when variable dbconnection is set to null, in the next line this object access its method `setAutoCommit(false)`. Since the object is null, the test output will the print the error trace. Dbconnection is set to null if `driver.connect(url)` returns null. The reason for this is that the URL provided during the `driver.connect` is an invalid one.

We also used a way to populate the possible list of driver name to test against the driver.connect method. The way we do this is by using the string automata of Symbolic PathFinder.

Listing 3.5: SPF example: Testing with strings

```
public boolean verifyDriver(String str) {
    int lastDot = str.lastIndexOf('.');
    if (lastDot < 0) {
      return false;
    }
    String rest = str.substring(lastDot + 1);
    if (!rest.contains("driver")) {
      return false;
    }
    if (!str.startsWith("com.")) {
      return false;
    }
    String t = str.substring("com.".length(), lastDot);
    if (t.startsWith("dummysite.")) {
          t = t.substring("dummysite.".length());
    }
    if (!(t.equals("mysql")) && !(t.equals("oracle"))) {
       return false;
    }
    return true;
}
```

A method is created to specify the possible combination of the string format to make it use. For the driver connection, rather we can make use of complete string than using method to get possible string outcomes. But this is a way to show that we can use any mechanism for the string operations. The generated output from this method is added to an array list in the `open()` method. The variable driverNameList will have list of string to test against the driver connection URL. This variable is driven by JPF's Verify. Verify getint method accept integer ranges. The range is equal to the number of string list in the `driverNameList`. Every time Verify getint enumerates it ranges with specific int value, respective string list is used for driver connection URL.

After investigating each of the tools and their advantages and limitations, we decided to use Symbolic PathFinder for our research in unit test isolation and bug fixing process. Symbolic PathFinder is powerful in such that it uses model checking and symbolic execution along with constraint solvers to generate concrete test inputs automatically. It exhaustively explores all the behaviors of the system and provides all the satisfiable path conditions. On the other hand, jMock provides an advantage of stubbing objects that are yet to be implemented. Expectations can be added to stubbed object where one can control different return values. However, increase in code size will lead to increase in test cases and expectations and thus increasing chances to make error. Similarly, Korat is a constraint based tool that uses imperative predicate. It requires users to manually create constraints for test input generations which Symbolic PathFinder does automatically. In next chapter, we present case studies that apply unit level bug detection and provides examples on how symbolic execution can be used to detect bug in early phase.

Chapter 4

Case Studies

In this chapter, we will focus on bug detection, unit level isolation and fix them using Symbolic PathFinder. We will start by testing its ability to detect bugs in small programs, namely student programs from an introductory programming course. Then we will move on to Tomcat project and show how incomplete fixes are detected. We will provide detailed application of Symbolic PathFinder on isolated buggy code and explain different ways to detect and fix bugs.

4.1 Bug Detection in Small Programs

To support our research we collected some of the programming assignments completed by the students at University of Nebraska at Omaha. These programming assignments were: Fibonacci and Perfect numbers. The first program, Fibonacci, takes an integer input from a user and outputs the Fibonacci number at the given index in the series. The second program takes an integer input from a user and checks that number is a perfect number. Both of these assignments were relatively simple and small. However, these programming assignments contained programming literals such as conditional

statements and loops which were very useful to conduct our research to observe the behavior of the program when involving model checking.

4.1.1 Built-in listeners

Symbolic PathFinder has different listener in order to process the method under test for model checking. Some of the listeners that are in effective use are SymbolicListener and SymbolicSequenceListener. Both the listeners use a solver to solve each path condition, logs the symbolic value that matches the condition, and stores other detail information of the method under test. The difference between these two is that SymbolicSequence Listener produces a JUnit output and this is also responsible when operating string parameters via use of automata. Whereas SymbolicListener produces HTML output and it is responsible for solving data types such as int, double, Boolean.

4.1.2 Custom Listener

As we see that both listeners have their own advantages, we decided to combine some of the features from both and create our custom listener. It is created by overriding most of the code structure that SequenceListener carry with feature that produces JUnit test. To do this, we made changes to SequenceListener by adding a class that captures method and path condition details, added a method that publishes JUnit using the captured details. We used EqualsBuilder's reflectionEquals[37, 1]. It is a feature provided by Apache Jakarta Commons. It makes use of Java reflection to determine if two objects are equal based on field by field comparison. We used it to provide proper assertion to the automated test input generation for JUnits. After setting up our custom listener, we implemented our custom listener in our test method.

4.1.2.1 Implementation of Custom Listener

We used Fibonacci program to try out our custom Listener. The test of was done for a method called fibCalc which:

- takes the index of the Fibonacci number series and
- outputs its respective Fibonacci number

The return type of fibCalc is set to double because the output may have large number for greater index. We took the following steps:

- At first we took a Fibonacci program which had minor bugs.
- Then we isolated the method fibCalc in a new file and added the necessary precondition to it.
- A driver was added to the file in order to run the method fibcalc via Symbolic PathFinder.
- A .jpf config file was created where the supporting configuration information was provided such as target class, classpath, debug mode, listeners.

We ran Symbolic PathFinder on the isolated file along with the configuration settings. Symbolic PathFinder did its analysis (ie symbolic model checking) and produced a list of possible test input parameters for the fibCalc. In addition, it also generated its respective output value after feeding in the test input parameters. Manual analysis is where comes to an action after this step. Developers now have to decide whether the test input parameters has produced the correct output and also needs to makes sure that it has satisfied the preconditions that were provided. If all the output results are correct then we are good to mark fibCalc to be error free.

However, if the developers come across any errors then he can review the symbolic expression for the input parameter that gave incorrect result and makes changes to the fibCalc accordingly.

4.1.3 Bug Identification

After successful run with SymbolicListener, we used our custom listener to run the Symbolic PathFinder once more. The custom listener produces JUnit test cases as an output. We extracted these JUnit test cases and copied over to a separate Java file. Our goal here is to test other remaining Fibonacci programming assignment submitted by the students. We ran this JUnit on other programs and found out that some of the program submitted by the students had minor bugs and others had some major bugs. The minor and major bugs were categorized by identifying the number of test cases failed using the JUnit test cases that we generated. We once more ran Symbolic PathFinder with SymbolicListener on the programs that failed to satisfy the JUnit. We extract the symbolic execution statements from the output where the test failed for its respective test input. This way we were able to point out the exact place in the program where a fix was supposed to be made.

As a result, we were now able to quickly test other programs for any bugs and provide students with valuable suggestion for fixes. The developers or professors benefit with JPF's Symbolic PathFinder in order to generate random test input to test program and provide fix solution quickly. Professors now won't have to be dependent on the same test cases and thus generate different test input by changing preconditions to the program. If students get guidance on the use of symbolic JPF for their programming assignments then there is a chance that they will figure out their error in early stage and help them improve their coding skill as well.

4.1.4 Applying symbolic execution

The code snippet of fibCalc method shown in Listing 4.1 was tested against Symbolic PathFinder. The logic is simple. Here, the input is the index number and the output is the Fibonacci number at that index. The Fibonacci number are 0, 1, 1, 2, 3, 5, 8, 13, 24, 37, and so on. If the `fib_num`, input parameter, is 6 then the output will be 8 and if the input is 0 then the output will have 0.

Listing 4.1: Sample Fibonacci program

```
@preconditions("fib_num>=0.0&&fib_num<=70.0")
public static double fibCalc(double fib_num) {
    double prev2 = 0;
    double prev1 = 1;
    double main = 0;
    double ctr = 2;
    if (fib_num > 1) {
        while (ctr <= fib_num) {
            main = prev1 + prev2;
            prev2 = prev1;
            prev1 = main;
            ctr++;
        }
    } else {
        main = fib_num;
    }
    return main;
}
```

Below is the configuration required to run symbolic execution over the test method `fibCalc`.

```
target=exercise.a16.fibonacci.Fibonacci
classpath = ${project}/demo/bin
symbolic.debug=true
symbolic.min_double=-8.0
symbolic.max_double=100.0
search.multiple_errors=true
symbolic.method=exercise.a16.fibonacci.Fibonacci.fibcalc(sym)
listener = gov.nasa.jpf.symbc.SymbolicListener
```

The configuration file requires the target class, classpath where the Java bytecode reside and other symbc parameters such as min max double range for test inputs, search for multiple errors, the target method under test, and listener. The debug mode set to true lets you view all the path condition evaluated in the output which help in analyzing the steps taken by symbc verification. Table 4.1 shows the summary of test results for one student.

fib_num	return
0.0	0.0
1.0	1.0
1.000001	0.0
2.0	1.0
2.000001	1.0
3.0	2.0
3.000001	2.0

Table 4.1: Method fibCalc input/output values

4.1.5 Result

The results in the table are only a part of output. Looking closely at the table, there is one incorrect output and that is when 1.000001 is given as input param, the result returned is 0.0 instead of 1.0. This is the only incorrect return value generated during symbc verification. After acknowledging this issue, one can quickly navigate the the fibCalc method and check for the cause of this error. Alternatively, the symbc verification result also provides the path condition as an output when debug mode is on and which proves to be very helpful in fixing bugs.

```
numeric PC: constraint # = 2
fib_num_1_SYMREAL <= CONST_70.0 &&
fib_num_1_SYMREAL >= CONST_0.0 -> true

### PCs: total:1 sat:1 unsat:0
```

```
numeric PC: constraint # = 3
fib_num_1_SYMREAL < CONST_1.0 &&
fib_num_1_SYMREAL <= CONST_70.0 &&
fib_num_1_SYMREAL >= CONST_0.0 -> true

*************Summary***************
PC is:constraint # = 3
fib_num_1_SYMREAL[0.0] < CONST_1.0 &&
fib_num_1_SYMREAL[0.0] <= CONST_70.0 &&
fib_num_1_SYMREAL[0.0] >= CONST_0.0
Return is:  fib_num_1_SYMREAL[0.0]
***********************************
```

The above constraints produced by Symbolic PathFinder for each path condition has its own verification levels where the result is either true or false. The number following the constraint # is the total number of constraints for a path condition. The first constraint results true for any symbolic parameter value `fib_num` that is greater or equal to zero and less than or equal to 70 which is the precondition for the method under test. A constraint is added as the path condition goes from one statement/condition to another. The second constraint with total of three constraint checks if symbolic parameter value is less than 1.0, an addition constraint to previous. This new constraint does not violate the precondition and hence results to true. Finally, the summary of the resultant constraint is displayed with values appropriate with the constraints that resulted true. The return value is displayed which is the result of inputs provided according to the constrain satisfied.

```
numeric PC: constraint # = 4
CONST_2.0 > fib_num_1_SYMREAL &&
fib_num_1_SYMREAL > CONST_1.0 &&
fib_num_1_SYMREAL <= CONST_70.0 &&
fib_num_1_SYMREAL >= CONST_0.0 -> true

*************Summary***************
PC is:constraint # = 4
```

```
CONST_2.0 > fib_num_1_SYMREAL[1.000001] &&
fib_num_1_SYMREAL[1.000001] > CONST_1.0 &&
fib_num_1_SYMREAL[1.000001] <= CONST_70.0 &&
fib_num_1_SYMREAL[1.000001] >= CONST_0.0
Return is:  CONST_0.0
**********************************
```

4.1.6 Bug Fixing

The path condition with four constraints above is similar to the path condition with three constraint. The precondition on both the path condition is same ie `fib_num >= 0 and <=70`. The other two conditions are that the `fib_num > 1.0 and fib_num < 2`. In the summary section of path condition with four constraints, the appropriate values are assigned symbolically meeting the constraint and the return value is displayed. As seen, the return value is 0.0 which is an incorrect output. The correct output for the constraint should have been 1.0. From both table and constraint, we can see that the return value by `fibCalc` is 0.0 for `fib_num > 1 and < 2`. Therefore, we would need to analyze the constraint and fix the program logic accordingly. There are two possible fix to this problem. The first is to change the data type of `fib_num` to int and the second fix would be to replace

`if(fib_num > 1)` to `if(fib_num >=2)`

Any one of these solutions will fix the problem and running Symbolic PathFinder over the modified program will provide a set of input symbolic values and its corresponding values which are all correct. However, the path condition and the symbolic values for input test generation result may vary.

We now have a program which gives the correct return values for every input parameter. Hereafter, custom listener can be used to run Symbolic PathFinder over the test method and the JUnit that we get from this run can be used to test Fibonacci

program written by other students. Later, the program which fails the JUnit test will have an advantage of which test input parameters failed. With the help of path condition and constraints will help the program logic to be fixed to have correct outputs. The limitation of our approach is that, we have tested the method with data types such as int, double, and Boolean. The string data type operation is quite tricky and requires extra configuration parameter while running Symbolic PathFinder. Apart from this, there is also a recommendation to the professors who assign the programming assignments to the students. On providing programming assignments, professors need to ask students to use their specified interface with definite signature. It's the best way to make the test method's signature even across every student which eventually will make symbolic test easier.

4.2 Detecting Incomplete Fixes in Tomcat

We explored the Bugzilla of Apache Tomcat server. Apache Tomcat [2] is an open source software implementation of the Java Servlet and JavaServer Pages technologies. We explored the bugs related to Apache Tomcat 6. Our primary target was to identify the reopened bug. Most of the reopened bugs were to do with runtime exception such as ArrayIndexOutofbound, NullPointer etc. Besides these, bugs related to logical error were also visible. The bug information from the Bugzilla provides the cause of the bug, bug traces, fix suggestion etc. Discussion on the bug information suggests insight on the fix and developer providing patches helps finding a way to solve it. Given these information, we used Apache Tomcat's version repository page to locate the file that had related bug in it. After getting the right file, we first studied the affected code segment from the file, extracted it and isolated it. After isolating the code segment, we apply Symbolic PathFinder to execute the code symbolically.

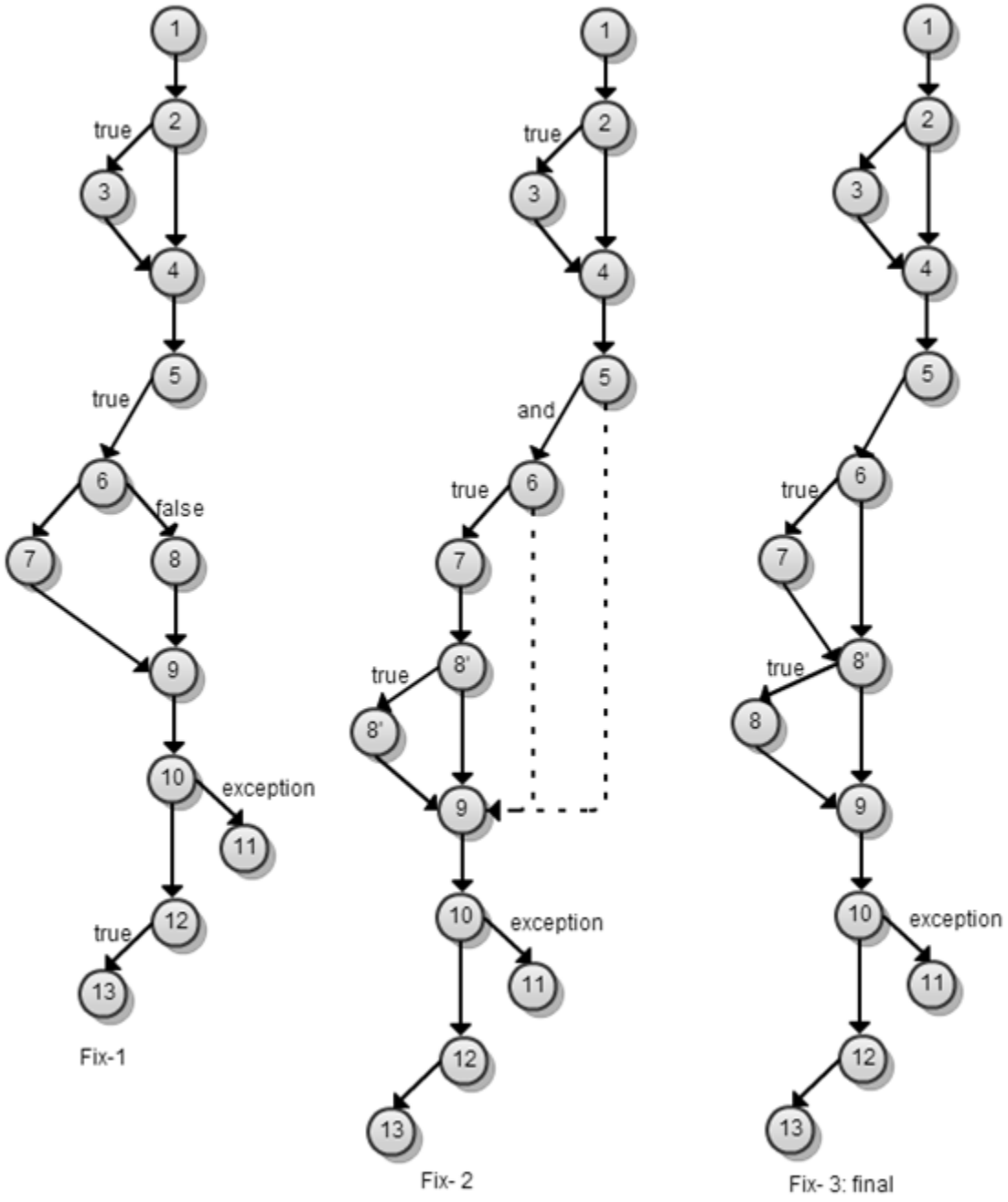


Figure 4.1: Code flow graphs

Executing the code symbolically gives the advantage of identifying the possible test generation for the code to execute. This way, we will be able to scrutinize the code segment and fix any regions where loop holes are identified. We also used Symbolic PathFinder's code coverage feature to track down the coverage covered by the code segment while execution. The generated test cases gave us a clear idea on how each code branches are getting covered with test input generated by Symbolic PathFinder.

Listing 4.2 shows a method named `report`. Figure 4.1 represents the overall flow of the method report for every attempt made to fix the code snippet. For Fix-1, line number 7: the value for message is supplied by throwable.getMessage(). The possibility is that throwable's getMessage can return null. If it returns null then the line number 10 have the report value formatted with a null value which is not appropriate. Thus, line number 7 is where the bug lies and we would need to extract only the parts of the method that actually contribute to the bug and make sure that bug is preserved.

Line 1, 2, and 3 can be ignored because they don't contribute to the cause for bug. Similarly Line 9 and beyond can also be ignored because further this line, the message value is only used and not assigned. Therefore, we extract lines between 4 to 8 inclusive and isolate them to recreate similar working mechanism and preserve the bug. Similarly, Fix-2 represents a fix to Fix-1 but it still has a bug. Whereas, Fix 3 is the final fix that resolves the issue. Three flow graphs in the code flow can be compared and distinguish the code flow and how it has affected the overall flow with the changes to the code.

Listing 4.2: Tomcat buggy code snippet (See Figure 4.1 Fix-1)

```
protected void report(Request request, Response response,
    Throwable throwable) {
1   int statusCode = response.getStatus();
2   if ((statusCode < 400) || (response.getContentWritten() > 0)) {
3       return;
```

```
    }
4   String message = RequestUtil.filter(response.getMessage());
5   if (message == null) {
6       if (throwable != null) {
7           message = RequestUtil.filter(throwable.getMessage());
        } else {
8           message = "";
        }
    }

9   String report = null;
    try {
10      report = sm.getString("http." + statusCode, message);
    } catch (Throwable t) {
11      ExceptionUtils.handleThrowable(t);
    }
12  if (report == null) {
13      return;
    }
}
```

After examining the control flow graph, we extract the sub-part of the method that actually contributes to buggy behavior. Isolating this sub-part of the method and running them guarantees that we have preserved the bug. Later, we construct isolated method, which will contain the extracted code, with symbolic variables using Symbolic PathFinder. The inputs for the method are set and injected at runtime. These inputs can be a range of value, comma separated string value which is used by symbolic variables when a method is enumerated every time with a different value. Some of these values are set in the configuration files and others (such as String values) can be set to a method that is used by the isolated method under test. These values can actually control the different path flow under test. The control flow graph can provide us with information as what values should be used or added to the configuration files or to the method that provides input. Restricting these values can help us focus only the paths that lead to code failure.

Listing 4.3 shows a code snippet from an isolated method. The two variables, `response` and `throwable`, represents symbolic reference. The symbolic value for

these variables are provide by a method called `getMessage()` in their own class.

Listing 4.3: Symbolic reference

```
Response = (Response) Debug.makeSymbolicRef("input_C", new Response());
throwable = (Throwable) Debug.makeSymbolicRef("input_D", new Throwable());
```

The `getMessage()` method shown in Listing 4.4 had a `String` array variable called messages which contains all the possible values that is returned when this method is invoked, one at a time. The value of the digit class variable is how the array index is determined.

Listing 4.4: getMessage() method in Response and Throwable class

```
public String getMessage() {
        String[] messages = new String[]{null,"something"};
                return messages[this.digit];
}
```

In Listing 4.5, the digit class variable of both Response and Throwable class are explicitly assigned to JPF's API – Verify. It is a data choice generator that is suitable for writing test drivers that are model check aware. It obtains non-deterministic input values from JPF in a way that it can systematically analyze all relevant choices. The method will be executed for both values 0 to 1.

Listing 4.5: Explicit use of JPF's API - Verify

```
if (response != null) {
    this.response.digit = Verify.getInt(0,1);
}
if (throwable !=null){
        this.throwable.digit = Verify.getInt(0,1);
}
```

Listing 4.6 is the original code that has a bug in it as stated earlier. The objective of this code snippet is to have a message variable either filled with a value or an empty string. However, if you closely look at the refined version of the code, there

is a bug. The code with the line `this.throwable.getMessage()` can return a `null` value and therefore the code is prune to bug and will result in logical failure.

Listing 4.6: Original code

```
if (message == null) {
    if (throwable != null) {
        message = RequestUtil.filter(throwable.getMessage());
    } else {
        message = "";
    }
}
```

Listing 4.7: Fix to previous bug

```
if (message == null && throwable != null) {
    message = throwable.getMessage();
  if (message == null) {
      message = "EMPTY";
    }
}
```

Listing 4.8: Fix to previous bug (See Figure 4.1 Fix-2)

```
protected void report(Request request, Response response,
                      Throwable throwable) {

1     int statusCode = response.getStatus();

2       if ((statusCode < 400) || (response.getContentWritten() > 0)) {
3           return;
        }

4       String message = response.getMessage();
5,6     if (message == null && throwable != null) {

7           message = throwable.getMessage();
8*          if (message == null) {
8               message = "EMPTY";
            }
        }

9       String report = null;
        try {
10          report = sm.getString("http." + statusCode, message);
        } catch (Throwable t) {
11          ExceptionUtils.handleThrowable(t);
        }
12      if (report == null) {
```

```
13              return;
            }
}
```

Listing 4.7 is the fix to the previous bug introduced in Listing 4.6 and Listing 4.8 is the overall method that includes the fix. The throwable check for null is moved one step above and a null check against message is added on line 8*. The check for message with null is an appropriate fix. However, line 5,6 has a still a bug that got introduced when trying to fix previous bug. If message and throwable both carries a null value in line 5,6 then the if condition will fail and the message still end up with a null value.

Listing 4.9: Final fix

```
if (message == null) {
    if (throwable != null) {
        message = throwable.getMessage();
    }
    if (message == null) {
        message = "EMPTY";
    }
}
```

Listing 4.10: Final fix to previous bug (See Figure 4.1 Fix-3)

```
protected void report(Request request, Response response,
                      Throwable throwable) {

1       int statusCode = response.getStatus();
2       if ((statusCode < 400) || (response.getContentWritten() > 0)) {
3           return;
        }

4       String message = response.getMessage();
5       if (message == null) {
6           if (throwable != null) {
7               message = throwable.getMessage();
            }
8*          if (message == null) {
8               message = "EMPTY";
            }
        }

9       String report = null;
```

```
        try {
10          report = sm.getString("http." + statusCode, message);
        } catch (Throwable t) {
11          ExceptionUtils.handleThrowable(t);
        }
12      if (report == null) {
13          return;
        }
}
```

Listing 4.9 is the fix to the previous bug introduced in Listing 4.7 and Listing 4.10 is the overall method that includes the fix. This fix does not contain any bug. The message variable ends up with either a definite value or an empty value. The code takes care of null values very well this time. It took three attempts to fix the working of code snippet. Had the developer used a tool that checks every possible value or instance that may occur for each of the variables then the logical bug would have been identified after the first attempt. We ensure that these problems can be easily identified with the use of Symbolic PathFinder. It identifies each branch and produce possible values that satisfies the condition and also the counter-example for the same.

Below are the results of using Symbolic PathFinder and using these results we can determine if the code segment under test is bug free.

```
***Execute symbolic INVOKEVIRTUAL: report()V  ( )
response: verification.string.Response@aaf5, throwable: null
message: null, throwable: null
result-> message: EMPTY
================================================
response: verification.string.Response@aaf5, throwable: null
message: something, throwable: null
result-> message: something
================================================
response: verification.string.Response@aaf5,
throwable: verification.string.Throwable@aa8f
message: null, throwable: verification.string.Throwable@aa8f
result-> message: null
```

```
================================================
response: verification.string.Response@aaf5,
throwable: verification.string.Throwable@aa8f
message: null, throwable: verification.string.Throwable@aa8f
result-> message: something
================================================
response: verification.string.Response@aaf5,
throwable: verification.string.Throwable@aa8f
message: something, throwable: verification.string.Throwable@aa8f
result-> message: something
================================================
response: verification.string.Response@aaf5,
throwable: verification.string.Throwable@aa8f
message: something, throwable: verification.string.Throwable@aa8f
result-> message: something
================================================
```

The above output was seen when we ran code 5 with SPF. In the result, we can clearly see that one of the message is null as a final result. The developers objective was to prevent message from having null value which eventually failed in this attempt. Similarly, when we ran SPF on code 4, we got the following result as below:

```
***Execute symbolic INVOKEVIRTUAL: report()V  ( )
response: verification.string.Response@aaf5, throwable: null
message: null, throwable: null
result-> message: EMPTY
================================================
response: verification.string.Response@aaf5, throwable: null
message: something, throwable: null
result-> message: something
================================================
response: verification.string.Response@aaf5,
throwable: verification.string.Throwable@aa8f
message: null, throwable: verification.string.Throwable@aa8f
result-> message: EMPTY
================================================
response: verification.string.Response@aaf5,
throwable: verification.string.Throwable@aa8f
message: null, throwable: verification.string.Throwable@aa8f
result-> message: something
```

```
================================================
response: verification.string.Response@aaf5,
throwable: verification.string.Throwable@aa8f
message: something, throwable: verification.string.Throwable@aa8f
result-> message: something
================================================
response: verification.string.Response@aaf5,
throwable: verification.string.Throwable@aa8f
message: something, throwable: verification.string.Throwable@aa8f
result-> message: something
```

The results are all in green which means the code didn't result in any null value for the message object. Hence, the code 7 is the final fix.

On the other hand, we tested the same code using jMock, a mocking framework used for Java. Listing 4.11 is how we set up the mock for Listing 4.3.

Listing 4.11: jMock version of stub

```
public void testErrorReportValue1(){
        Mockery context = new Mockery();
        final Response mockResponse = context.mock(Response.class);
        final Throwable mockThrowable = context.mock(Throwable.class);

        context.checking(new Expectations(){{
                oneOf(mockResponse).getMessage();
                will(returnValue("something"));

                oneOf(mockThrowable).getMessage();
                will(returnValue("something"));
        }});
    ErrorReportValueBefore1 errorReport = new ErrorReportValueBefore1();
    String message = errorReport.report(mockResponse, mockThrowable);
    assertNotNull(message);
}
```

As seen above, it represents one of the several test-cases that was created using jMock. Every test feeds each of the possible returns values for the getMessage() method that is invoked by throwable and response. The values are used by the method under test. In the test, assert is used to check the returned message value not to be null. The test passes if the assert added is true. However, this process

requires developers to speculate possible values to be returned for each of the invoked method and therefore makes it difficult, time consuming and error prone. Suppose the method under test uses one more getMessage call then one more expectation is required to be added. Instead of returning two possible values for getMessage, it will require three different values and the combination will lead to couple of more test-cases. Unlike mock, Symbolic PathFinder does not require manual addition of combinations.

4.3 Preliminary Empirical Study

We showed two bug identification and fixes on single files. We were able to find latent bugs and fix them by the help of symbolic execution. In our research, we focused on fixing single files rather than multiple files fixes. The advantage is that these files will have less dependencies to manage and will have simpler forms. An important question at this point is, how often do such incomplete fixes occur at the unit level? As it turned out, it was difficult to find incomplete fixes by examining the bug repository and commit history. We first looked for bugs that were reopened, but most reopened bugs had been previously close with no fixes. Next, we looked at the commit history for each file to find if the same bug ireport identifier is mentioned in multiple commits but we also did not find many.

We then first conducted some baseline study on the number of commits that are unit-level, in this case, single-file commits. We downloaded all the changes committed for this project from its version repository. We created a tool to analyze the commit history and identify all commits that are confined to a single file, after eliminating non-Java files as well as test files. To identify the fixes, we analyzed the commit descriptions to looked for mentions of “bug” or “fix.” Table 4.2 shows our results.

Commit Types	Number of Commits	Percentage
All commits	9069	100%
Single file commits	3872	42.6%
All file fixes commits	2220	24.4%
Single file fixes commits	1164	12.8%

Table 4.2: Apache Tomcat - Single file fixes and commits

Commit Types	Number of Commits	Percentage
All commits	6359	100%
Single file commits	3029	47.6%
All file fixes commits	475	7.4%
Single file fixes commits	220	3.45%

Table 4.3: BioJava - Single file fixes and commits

Single-file commits comprise nearly half of all commits. Moreover, single-file fixes comprise half of all fixes. To check if this is an occurence that is unique to Apache Tomcat, we also tried it on a completely unrelated project, BioJava. Table 4.3 shows the results. The results from BioJava are consistent with the Tomcat results.

Though we cannot conclude that incomplete fixes occur quite frequently, these preliminary results look promising in that among the single-file fixes, there may be many incomplete fixes. As part of future work, we plan a more in-depth search for incomplete fixes. This requires the incorporation of a fault localization tool in order to determine if recurring commits to a file are occuring within the localized area where the fault is known to lie.

4.4 Discussion of Findings

Fixing bug and testing is a time consuming task and sometimes some fix does not seem correct. Researchers have been trying to study on automated test generation techniques and created several tools to make bug fixing easier and efficient. Although

these tools promise an approach to fix bug in an efficient manner, it certainly has limitation. We studied different methods that have been applied by several researchers. We also tested tools and techniques applied by them. Hence, our approach is to utilize some of the tools created by these researchers and try to use them in an effective way so that bug fixing process becomes easy and ensure code coverage with test inputs as well.

There are benefits of using symbolic execution over black box testing. Symbolic execution smarter and provides higher path coverage than black box testing. However, use of symbolic execution can be more complex compared to black box which is simple, lightweight, easy and fast. We do not suggest using only symbolic execution on the program to find bugs but also use the black box testing. It depends upon the application and the method under test on which test should be applied. It is always good to start with black box and remove the low hanging bugs. Later white box (symbolic execution) can be used to search for bugs that black box failed to identify. In our experiment, we have used Symbolic PathFinder which symbolically execute a method under test to find bugs. Apart from this, we have also used jMock on top of JUnit to mock real time data and return expectations. For methods that do not have conditional statements (branch), we can just make use of jMock and there is no need to use model checking. But for those with conditional branch, symbolic model checking should be used.

When using symbolic execution with Symbolic PathFinder, it would be useful to follow these steps:

- Always stub the object for any dependencies
- Apply pre-condition and post-condition oracles when applicable. Pre-conditions acts as a predicate to prune away unnecessary inputs

- If any variable/object needs to be executed symbolically, mark them as symbolic by using a annotation or using makesymbolicref method call.
- Turn off exception handled by users
- For string inputs, run the method with string input first, collect the results and add it to JPF's verify.getInt(n, m). This way all the string input will be tested for any errors.

Chapter 5

Conclusion and Future Work

5.1 Discussion

Software testing and maintenance has always played a major role in securing software quality, integrity and efficiency in the software development. There are uncountable tools in the market that you can use to check software for security loop holes to maintain software quality. Some of these tools are Sonar, Klocwork. Every software product goes through unit testing, integration testing, acceptance testing etc. and these take time and effort. JUnit, Selenium etc are the ones that are used for integration and unit testing to find bugs in the software. However, lack of expertise in creating JUnit and Selenium tests may miss out important errors.

Therefore we use Symbolic PathFinder which is based on symbolic execution that performs model checking and automatically generates test inputs with the help of constraint solvers. We demonstrated that use of symbolic execution is useful to detect latent errors and it provides us with detail information of each of the path condition. Analyzing the path condition we were able to understand the bug and suggest appropriate fix to it. We lay the groundwork for the creation of an automated

tool for isolating code by studying its feasibility and investigating existing testing technologies that can facilitate the creation of such drivers and stubs. We also studied use of: jMock, Korat and came to realize that they have some limitation which was overcame by Symbolic PathFinder. We found that almost half of the fixes were single file commits and thus application of symbolic execution over single files looks promising though inconclusive.

5.2 Limitations

We have only tested the programs with integer, Boolean and double data types for input parameters. Symbolic PathFinder does process string data type as symbolic. It uses automata for string test input generation. However, the generated string test input depends upon the string class libraries that compute on strings. The method under test with parameter as string data type should have string operation such as startsWith, endsWith, indexOf, contains, regex etc to filter or test the string in order to have desired content in the string literal. Symbolic PathFinder analyzes this string operation and generates list of test input according to it. We have implemented the string data type and tested it with the method under test. However, more can be done on it.

5.3 Future Work

In the future, we will focus on implementing symbolic execution on complex data structures to test its efficiency. Analyzing race condition extensively for programs using multiple threads could be another important area where this method can be used.

In our research, we have manually located using a bug report and isolated it. However, we have studied how bug localization can be automated and therefore we will work on developing a localization tool and incorporate an algorithm that can efficiently locate the bug automatically. We also plan an in-depth empirical study to identify incomplete fixes using a bug localization algorithm to identify the buggy code and identifying recurring commits that affected the code.

Bibliography

[1] Apache commons, equalsbuilder: http://commons.apache.org/proper/commons-lang/javadocs/api-2.6/org/apache/commons/lang/builder/equalsbuilder.html.

[2] Apache. http://tomcat.apache.org/.

[3] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 1–3, New York, NY, USA, 2002. ACM.

[4] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 326–335, Washington, DC, USA, 2004. IEEE Computer Society.

[5] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, H. Zheng, and Robby. Bandera: Extracting finite-state models from Java source code. In *Proc. of the Intl. Conference on Software Engineering (ICSE 2000)*, pages 439–448, 2000.

[6] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[7] C. DeMartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent java programs. *Software: Practice and Experience*, 29(7):577–603, June 1999.

[8] E. W. Dijkstra. *Notes on structured programming.* Technological University Eindhoven Netherlands, 1970.

[9] S. G. Eick, J. L. Steffen, and E. E. Sumner, Jr. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11), November 1992.

[10] C. Flanagan and S. Qadeer. Assume-guarantee model checking. Technical report, Microsoft Research, 2003.

[11] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 225–234, 2010.

[12] P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM.

[13] Z. Gu, E. Barr, D. Hamilton, and Z. Su. Has the bug really been fixed? In *Proc. of the Intl. Conference on Software Engineering (ICSE 2010)*, pages 55–64, May 2010.

[14] J. Halleux and N. Tillmann. Moles: Tool-assisted environment isolation with closures. In J. Vitek, editor, *Objects, Models, Components, Patterns*, volume 6141 of

Lecture Notes in Computer Science, pages 253–270. Springer Berlin Heidelberg, 2010.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, 2002.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Model Checking Software*, pages 235–239. Springer, 2003.

[17] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[18] IDC. A telecom and networks market intelligence rm. http://www.idc.com.

[19] jMock. http://www.jmock.org/.

[20] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization for fault localization. In *in Proceedings of ICSE 2001 Workshop on Software Visualization*, pages 71–75, 2001.

[21] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering*, 11(4):403–434, Oct. 2004.

[22] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 553–568, 2003.

[23] S. Korat. http://korat.sourceforge.net/.

[24] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. *SIGSOFT Softw. Eng. Notes*, 30(5):286–295, Sept. 2005.

[25] K. L. McMillan. The SMV system. In *Symbolic Model Checking*, pages 61–85. Springer, 1993.

[26] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat: A tool for generating structurally complex test inputs. In *Proc. of the Intl. Conference on Software Engineering (ICSE 2007)*, pages 771–774, May 2007.

[27] K. Muşlu, B. Soran, and J. Wuttke. Finding bugs by isolating unit tests. In *Proc. of the Intl. Symposium on the Foundations of Software Engineering (FSE 2011)*, pages 496–499, 2011.

[28] NASA. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc.

[29] T. Noll and B. Schlich. Delayed nondeterminism in model checking embedded systems assembly code. In *Proceedings of the 3rd international Haifa verification conference on Hardware and software: verification and testing*, HVC'07, pages 185–201, Berlin, Heidelberg, 2008. Springer-Verlag.

[30] C. Pasareanu and N. Rungta. Symbolic PathFinder: Symbolic execution of Java bytecode. In *Proc. of the IEEE/ACM Intl. Conference on Automated Software Engineering (ASE 2010)*, Sept. 2010.

[31] C. S. Păsăreanu, M. B. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *Theoretical and Practical Aspects of SPIN Model Checking*, pages 168–183. Springer, 1999.

[32] M. E. Scully. Korat Tester: An Eclipse plug-in for state space testing. Master's thesis, University of Texas at Austin, 2007.

[33] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto. Studying re-opened bugs in open source software. *Empirical Software Engineering*, pages 1–38, September 2012.

[34] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *Proc. Second Int'l Conf. Tests and Proofs*, pages 134–153, Apr 2008.

[35] W. Visser, K. Havelund, G. Brat, S.-J. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2), April 2003.

[36] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. of the Intl. Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 97–107, 2004.

[37] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, pages 380–403, 2006.

[38] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 14–24, 2012.

[39] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy. Characterizing and predicting which bugs get reopened. In *International Conference on Software Engineering (ICSE)*, pages 1074–1083. IEEE, 2012.