AUTOMATED STUDENT CODE ASSESSMENT WITH SYMBOLIC EXECUTION AND JAVA PATHFINDER

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Karl Bell

December 2012
COMMITTEE MEMBERSHIP

TITLE: Automated Student Code Assessment with Symbolic Execution and Java PathFinder

AUTHOR: Karl Bell

DATE SUBMITTED: December 2012

COMMITTEE CHAIR: Dr. John Clements

COMMITTEE MEMBER: Dr. David Janzen

COMMITTEE MEMBER: Dr. Gene Fisher
Abstract

Automated Student Code Assessment with Symbolic Execution and Java PathFinder

by

Karl Bell

The assessment of student code is a necessary part of most programming courses. However, many ways of assessing the correctness of student code can be very time-consuming and may be error-prone. This paper presents JSymTester, a tool which uses the symbolic execution framework of the Java PathFinder to find test inputs for student code and uses these inputs to extensively compare its functionality to a reference implementation. This allows for automatic testing of student code, relying only on the reference implementation and the student’s own implementation, eliminating the need to manually write tests. This tool was tested on small assignments for an introductory computer science course, and performed similarly to the existing, more traditional approaches of unit testing and output comparison. This shows that automated test generation techniques may, in general, be useful in the area of student code assessment.
Contents

List of Tables vii
List of Figures viii
1 Introduction 1
2 Test Generation 3
  2.1 Symbolic Execution .............................. 4
  2.2 Concolic Execution .............................. 7
3 Related Work 11
  3.1 Automated Test Generation in Java ................. 11
    3.1.1 Tools for Automated Test Generation .......... 11
    3.1.2 Concolic Execution in Java .................. 13
  3.2 Automated Assessment ................................ 14
    3.2.1 Tools for Automated Assessment ............... 14
  3.3 Test Data Generation in Education .................. 15
4 Implementation 16
  4.1 JPF Integration ................................... 17
    4.1.1 Adding a main() Method ...................... 18
  4.2 Object Reconstruction ............................. 18
  4.3 SymTestRunner ................................... 19
  4.4 Web-IDE Evaluator ............................... 20
5 Validation 21
  5.1 Test Setup ...................................... 21
5.2 Results ................................................................. 22

6 Future Work ......................................................... 26
   6.1 Beyond Code Coverage .......................... 26
   6.2 Language Feature Support .................. 28
   6.3 Ease of Use .............................................. 29
   6.4 Model-Checking ................................. 29
   6.5 Constructive Feedback ........................ 30

7 Conclusion ......................................................... 31

Bibliography .......................................................... 33
List of Tables

5.1 Overall test results .................................................. 22
5.2 Test results by exercise .............................................. 23
5.3 Failures not caught by evaluators ................................. 24
List of Figures

2.1 Example code and its symbolic execution tree ......................... 6
2.2 Example code and a concolic execution, step 1 ....................... 8
2.3 Example code and a concolic execution, step 2 ....................... 8
2.4 Example code and a concolic execution, step 3 ....................... 9
2.5 A code path which concolic testing can execute ..................... 9
Chapter 1

Introduction

The assessment of student code is a necessary part of most programming courses, as it provides a way to see if learning goals are being met. It directs the learning process of the student and helps them to see what they need to spend more time on. Unfortunately, assessing student code manually can be difficult and time-consuming. In addition, it is difficult to judge the correctness of student code without spending a large amount of time understanding it.[14]

For this reason, many instructors have seen fit to create their own automated testing tools to help assess student code, the majority of which have focused on examining student code functionality in some way. These tools can come in many forms, from scripting and output comparison, to using testing frameworks like XUnit.[14]

One problem with these approaches is that an instructor must write a suite of tests (or sets of inputs and outputs) to validate that the student’s code performs as expected in all cases. Not only can this take a significant amount of time, it is possible that the generated inputs will not take into account the oddities of
student code, as they are generally written before the student submits their code for evaluation.[14]

One way to address this issue is to write assignments in such a way as to get the students to write their own tests. This is the case for test-driven learning approaches.[4] However, this leads to the problem of verifying that the student written tests actually reflect the results which the instructor desired, essentially requiring at least some instructor testing after all.

This thesis presents the “JSymTester” tool for the Java programming language, which seeks to solve these problems by providing a way of automatically assessing student code given only a reference implementation of a programming exercise. It utilizes automated test generation techniques on this reference implementation as well as the student’s implementation to develop a suite of inputs which tests student code fully, using the instructor code as a test oracle.

JSymTester is based on the symbolic execution module for the Java PathFinder[23], and can be used independently as a command-line application, or as as an evaluator for the WebIDE platform[4].

Section 2 of this thesis provides background information about test generation. Section 3 looks at some of the related work in automated test generation and automated student code assessment. Section 4 goes into detail about how the JSymTester works, and Section 5 describes the procedures for and the results of our evaluation of it. Section 6 provides a number of suggestions for future improvements to the tool and identifies other related avenues of possible research, and Section 7 offers concluding remarks.
Chapter 2

Test Generation

To understand the implementation of JSymTester, it makes sense first to examine automated test generation in general. The Test Generation problem can be stated as follows[6]:

Given a sequential program with a set of input parameters, generate a set of test inputs that exercises as many program statements as possible.

There has been a large amount of work in the area of automated test generation. In general, the goal of all of these is, as stated above, to maximize code coverage. This is generally done with some kind of input space exploration, in order to find inputs which cause particular paths to be executed. A simple way of finding test inputs is to randomly generate them. Indeed, this the is the concept behind blackbox fuzzing, which throws random inputs (or randomly mutated correct inputs) at a program to see how it behaves.[10]

A more guided approach to this state space exploration can be taken by taking into account the structure of the program when generating test inputs. Rather than simply generating random inputs, one can use some knowledge of
the program’s internals to generate test inputs which are likely to get the program follow different code paths. This is generally called whitebox testing, and comes in many forms.[10]

The automated test generation tools which are most relevant to this thesis are based on a form of whitebox testing called dynamic test generation or concolic execution, which is itself an extension of a technique called symbolic execution. A number of tools exist which implement this test-generation strategy.[21, 9, 22] One of these tools is the symbolic execution module for the Java PathFinder, which was used to construct the JSymTester application described here.[23]

It is useful to note that, traditionally, all of the path exploration techniques described above are accompanied by some kind of model-checking to ensure correctness, as there is no test oracle available.[23] In the case of this thesis, the lack of expected outputs is not a problem, as we have a test oracle in the form of the instructor’s reference implementation.

2.1 Symbolic Execution

Symbolic execution is a type of program analysis which is based on running a program with symbolic inputs rather than real ones.[18] This means that variables referred to in the program, rather than being given a real value (such as an integer) are instead symbolic expressions based on the symbolic inputs to the program.

For example, we might have a program which takes an input \( x \) and returns \( x + 5 \). With regular concrete execution, we would give \( x \) an actual value (say, 3), and we would observe the return value to be a an actual value (say, 8). With symbolic execution, we instead substitute a symbolic value for \( x \) (say, \( a \)). We
would then express the return value as $a + 5$. The same holds for any return values or variables in a given program, in that they would all be expressed in terms of the symbolic inputs (in addition to hard-coded values like 5).

However, just treating inputs as symbolic tends to give us little information, unless it also takes into account the branch points of the program. Doing so allows symbolic execution to be used to analyze the possible paths of a program.[23]

Each possible code path has logical expressions which must be true in order for that code path to be followed. Since we are keeping track of all variables in terms of the symbolic inputs, these conditions can also be expressed in these terms. Each possible path through the program yields a set of logical expressions that are called that path’s path constraints. These constraints are expressed in terms of symbolic input to the program, and can be solved to generate a test input that would cause that particular path to be executed.[23]

Figure 2.1 shows a bit of example code along with its symbolic execution tree. Each possible path down this tree represents a code path, and the relevant statements (conditionals and assignments) would be kept track of by a symbolic execution engine. Assignments must be tracked down the tree for later conditions to depend on.
if (x == 5) {
    if (y > 0) {
        return x * y;
    } else {
        x = y;
    }
} else if (y < 0) {
    return y;
}
return x;

Figure 2.1: Example code and its symbolic execution tree.

The usefulness of symbolic execution for test input generation is apparent: Symbolic execution provides sets of logical constraints for all possible paths, and each of these sets of constraints can be solved to find a test input which covers a different code path.

However, symbolic execution does have some problems. First, it is limited by the constraint solver used. If a set of path constraints can’t be solved by the constraint solver, then the symbolic execution engine can’t generate inputs which would cause that path to be executed. This can happen frequently when there is some complicated math going on, such as with hash functions, which are deliberately hard to reverse.[7]

Second, symbolic execution has a hard time modeling system calls and non-deterministic functions.[7] In both of these cases, the outputs of these functions have unknown constraints. Their behavior may modify the code path followed in un-predictable ways, and branches based on their results are more or less impossible to enter unless the methods can be abstracted out or controlled by a
2.2 Concolic Execution

Concolic testing, or dynamic test generation, extends symbolic execution by interleaving concrete execution so that portions of a program which are difficult to reason about symbolically (such as hash functions) can be abstracted out by using actual values in these portions, instead of using symbolic ones.[6]

Concolic testing works by running through the program using concrete inputs, collecting the path constraints (again, in terms of the symbolic inputs) which are true of those inputs as the program runs. The first set of inputs is usually either randomly generated or set to some common or ordinary values. Once the first run is completed, the path constraints for that run are known. The last of these constraints is then falsified, generating a new set of inputs which can be fed into the process again. This process continues until there are no more solvable paths left to explore.[20]

This process is illustrated in Figures 2.2 through 2.4, using the example code from Figure 2.1. In Figure 2.2, the code is run using the shown inputs. This causes the code to run through the path highlighted in green. The path constraints are collected as they are encountered, resulting in the list shown as “PC.”
\[ x = 0, \ y = 0 \]

\begin{verbatim}
if (x == 5) {
    if (y > 0) {
        return x * y;
    } else {
        x = y;
    }
} else if (y < 0) {
    return y;
}
return x;
\end{verbatim}

PC: \((x != 5, y >= 0), \text{ret} == x\)

**Figure 2.2: Example code and a concolic execution, step 1.**

In Figure 2.3, the last condition from the previous step is falsified, resulting in new input values. The code is run through again to gather the path conditions for this path.

\[ x = 0, \ y = -2 \]

\begin{verbatim}
if (x == 5) {
    if (y > 0) {
        return x * y;
    } else {
        x = y;
    }
} else if (y < 0) {
    return y;
}
return x;
\end{verbatim}

PC: \((x != 5, y < 0), \text{ret} == y\)

**Figure 2.3: Example code and a concolic execution, step 2.**
In Figure 2.4, since no new conditions are added, the last previously unfalsified condition is falsified to get new inputs. This is then run through the code to get a new path constraint. This process continues until all paths which can be solved for have been explored (this has not been illustrated, for the sake of brevity).

\[
x = 5, \ y = -2
\]

```c
if (x == 5) {
    if (y > 0) {
        return x * y;
    } else {
        x = y;
    }
} else if (y < 0) {
    return y;
}
return x;
```

PC: (x == 5, y <= 0), ret == y

![Figure 2.4: Example code and a concolic execution, step 3.](image)

Concolic testing improves on symbolic execution in two ways. First, it guarantees that any inputs generated do, in fact, cause the expected path to be followed. After all, it has a concrete run of the program to prove this. Second, it allows the program to execute paths which may be impossible to run using just symbolic execution.[6] To illustrate this, imagine a program like the one in Figure 2.2.

```c
if (x == hash(y)) {
    // do something interesting
}
```

![Figure 2.5: A code path which concolic testing can execute](image)
The hash function in this example is likely something which it is difficult to reason about symbolically.[6] Doing so would constitute reversing the hash, something which the hash would likely be designed to prevent. The conditions it introduced on the input would likely be unsolvable by most constraint solvers (at least in a reasonable time). However, since concolic execution will have recorded the actual result of the hash call, it will know what a given y value hashes to. It can then set x to this value while keeping y stable, thus solving the condition for this branch.

It is also worthwhile to note that this is an advantage of concolic execution over standard blackbox fuzzing, which would just throw random values at this for a while. Such a process is very unlikely to hit the condition where x is equal to the hash of y, though concolic execution can handle this case quite easily.[6]

There have been numerous advances in the area of dynamic test generation in the past few years which have increased its effectiveness. Many new strategies and algorithms have made the technique much faster and more able to target relevant areas of code.[6, 1, 18, 12]
Chapter 3

Related Work

There is a large body of work in the fields of test generation and specifically symbolic execution, as well as in automated assessment. However, there has been limited work in the combining of these two fields.

3.1 Automated Test Generation in Java

There are a number of frameworks that have been created for automated test generation in the Java programming language. These tools incorporate a number of different test generation techniques, which, while they are not used by the J Sym Tester, may also be applicable in the realm of student assessment.

3.1.1 Tools for Automated Test Generation

Many of the tools available for test generation use some variation of random test generation, which involves throwing random inputs at programs to generate interesting behavior. Randoop is one of these.[17] It implements a technique
called feedback-directed random testing, which uses execution feedback gathered from executing test inputs as they are created in order to avoid generating redundant and illegal inputs. It does this by exploring the method sequences space for objects based on a specified contract. Those method sequences which cause a violation of the contract are marked as such, and sequences do not are saved as regression tests.[17]

Another category of test-generation tools use a technique called evolutionary testing. This uses evolutionary algorithms to evaluate the “fitness” of test inputs based off of particular criteria such as statement coverage, branch coverage, or size of generated tests. These inputs are then mutated in an attempt to derive a more “fit” set of inputs. This technique can be used either on the method level to explore method inputs, or at the class level to find method call sequences. Two tools which implement this approach are TestFul and EvoSuite.[2, 5]

Many test-generation tools focus on the specification of contracts for methods or objects which can be used to generate tests. One such tool is Korat, which uses formal specifications for methods to generate tests. Korat takes these formal specification in the form of method pre- and post-conditions. Korat generates test inputs by exploring the space of inputs which satisfy these pre-conditions to generate a set of inputs, then determines whether the post-conditions are met by the run of the method. This allows for generating valid failing tests in the absence of a test oracle.[3]

A somewhat different, but related field of research is that of test case testing, which aims to determine the quality of a given test suite. One of the popular ways of doing so is using mutation testing, which mutates particular (or random) parts of the source code which is meant to be tested, in order to see if this mutation breaks the tests. Javalanche and Jester are two examples of tools using
this technique.[16, 19] While this is not directly related to this thesis, it may be useful to use in the future to determine how well the JSymTester functions by using it to compare a randomly mutated reference implementation to the original.

3.1.2 Concolic Execution in Java

While these tools are all useful, this thesis focuses on the use of symbolic and concolic execution in Java, for which there are a few frameworks.[20, 23, 24] The two most well-known of these are jCUTE and the Java PathFinder.

jCUTE

jCUTE was developed at the University of Illinois at Urbana-Champaign, and provides a concolic unit testing engine for Java.[20] It is based on CUTE for C, and works by tracking values in memory and constraints on them to implement Concolic execution. It does not appear to be under active development, and the source was not readily available.

Java PathFinder

JSymTester utilizes the Java PathFinder, an open source tool maintained by the NASA Ames Research Center which is currently under active development.[23] The Java PathFinder (JPF) functions by providing a reimplementation of the Java Virtual Machine in Java which allows developers using the framework to instrument the running of Java programs at the bytecode level. Java PathFinder includes a module for symbolic and concolic test generation based on this bytecode instrumentation. This framework was used for this project.
3.2 Automated Assessment

A useful survey of automated assessment tools was performed by Ihantola et al in 2010.[14] This survey described trends among a number of different automated testing tools, and found that most automated assessment is being done by assessing functionality (that is, the behavior of the program), rather than using other analytic tools to assess style or performance. The survey listed the different approaches which had been taken to the problem of analyzing student code, most of which depended on output comparison, or unit testing (though often in combination with scripts or other testing frameworks).[14]

One of the key problems identified by this survey is that most assessment tools are written as one-off programs, usually just for the purposes of one class or one assignment. Many instructors write their own frameworks, though there are some more generalized tools which are widely known. This reveals a problem in that most tools are not written to be generally applicable and easy to use, though some attempts have been made.[14]

3.2.1 Tools for Automated Assessment

One such automated assessment tool is Web-CAT, which tries to get students to generate tests for their own code, but also provides tools for instructors to automate other testing of student code.[4] Since Web-CAT is extensible, it may be worthwhile to attempt to adapt the JSymTester to it in the future.

As mentioned above, one-off scripts or suites of unit tests are also used as automated assessment tools. These have the advantage of being generally easy to write and run, but rely solely on some form of output comparison in order to
function. This means both that all those outputs must be determined beforehand, and that they must be determined by hand or with a reference implementation. Not only can this take a significant amount of time, it is possible that the generated inputs will not take into account the oddities of student code, as they are generally generated before the student submits their code for evaluation, or at least without direct knowledge of the structure of all submitted student programs.

WebIDE, a teaching platform developed at Cal Poly, also incorporates automated assessment of student code correctness. It works by sending code to “evaluators” which are implemented as web servers which compile and run this code, testing with whatever unit tests the instructor provides. There are also evaluators for standard output comparison. The WebIDE evaluator framework provides a relatively simple way to express the tests which must be performed on student code, and does not necessarily entail use of the student’s code in generation of these tests.[4]

3.3 Test Data Generation in Education

A similar form of test input generation for assessing student submissions has been tried once before.[13] This tool was built using an early version of JPF’s symbolic testing framework which was not publicly available at the time. As such, the tool was not made available. In addition, the symbolic framework of JPF has since changed a large amount, meaning the work done in [13] would no longer be compatible with the current JPF codebase.

While the tool was somewhat related to JSymTester, the paper itself focused more on strategies for test generation and use of the JPF’s model checking techniques, rather than on the creation of a tool based solely on symbolic execution.
Chapter 4

Implementation

JSymTester is based on the symbolic execution module for the Java PathFinder, and can be used independently as a command-line application, or as an evaluator for the WebIDE platform. In general, it takes two classes (one specified as the reference class, and one as the test class), and the name of one method to test. It then performs concolic test generation on the reference class and test class to get a set of inputs. It runs these inputs through the reference implementation to get expected results, then runs the same inputs through the test implementation, comparing the results. Finally, it outputs results, as a list of successful and/or failed inputs. A more detailed description follows.

1. Load test and reference classes
2. Check for the test method on each class
3. Add or replace the main() method for each class via bytecode manipulation
4. Run JPF Concolic Execution on the reference implementation
5. Run JPF Concolic Execution on the test implementation
6. Rebuild a list of inputs from the results of the two above runs, reconstructing primitives and objects as necessary

7. For each input, run both the reference implementation and the test implementation, recording differences

8. Output differences, if any were recorded

4.1 JPF Integration

In order to do concolic execution of Java code, JSymTester utilizes the Java PathFinder, described above.

JSymTester does this by starting up a JPF JVM using the symbolic execution framework’s bytecode instrumentation. It specifies the reference class and the method to test and a Listener object (based on the JPF symbolic framework’s provided SymbolicListener) to watch the execution. The JPF begins execution using the main() method of the specified class, and the listener waits for execution to enter the method being tested. Once it enters that method, the listener waits for that method to return, at which point it records the path constraints of that run through the method, and the values of the inputs which satisfy these path constraints. The symbolic execution framework will then negate some part of this condition and run through the method again, and, again, the listener will record the path constraints at return time. This happens until there are no more negatable path conditions.

The Listener has then built up a list of path conditions and their solutions, which can be accessed by the JSymTester. This same thing is done again for the test version of the class, getting another set of inputs.
4.1.1 Adding a main() Method

One problem with the above sequence of events is that the JPF must have some point of entry to even begin execution at all. This means that any class which is being tested must have a main() method. This is a limitation of the JPF which is rather inconvenient if you wish to use it to test only a small function which doesn’t necessarily have to run as a whole program.

Rather than forcing instructors to ensure that both their and their students’ code have main() methods in them, JSymTester automatically adds a main() method for them using bytecode manipulation.

Before beginning JPF execution, the JSymTester uses reflection to inspect the class being tested for a main() method. If a main() method does exist, it is removed. The JSymTester then adds a main() method to the object’s bytecode using Javassist, a bytecode manipulation framework. This can then serve as the entry point for the JPF, without forcing instructors or students to write main() methods or ensure that their main() methods call the method they wish to test.

4.2 Object Reconstruction

Once the JSymTester has a list of inputs which each represent a different code path, it has to actually run the reference class with these inputs to generate the expected results. The solutions which are provided by the JPF’s symbolic execution are in the form of classes used by that code (SymbolicIntegers and SymbolicReals), which can’t be passed directly to the method under test. However, these classes do maintain a concrete representation of these values (as is required by concolic testing).
In order to actually call the method, JSymTester must convert inputs from this form to actual Java primitives or Java objects. For primitives, this is relatively simple, as byte, short, char, int, and long data types can easily be converted from the concrete value stored in SymbolicIntegers, and float and double types can easily be retrieved from SymbolicReals. In addition, booleans also are not much work, as they are stored as SymbolicIntegers of value 0 or 1, which are easily converted.

The problem arises when the method being tested takes some arbitrary object as an input. The JPF symbolic execution framework supports this by using lazy initialization.[15] The output of this is a number of integers and reals which represent the parent object and its fields (and fields of any field objects, recursively). This must be reconstructed into the actual object in order to run the actual implementation.

To reconstruct these objects, JSymTester constructs a key-value map to represent each input object. The keys are the field names, and the values are the values of these fields (either as a Java primitive or as a another map representing another object). This map can then be used, along with reflection, the rebuild the object to be passed in. JSymTester does this using the ObjectMapper provided by the Jackson JSON library.

4.3 SymTestRunner

The command-line version of the tool works as follows:

SymTestRunner [-v] REFCCLASS TESTCLASS METHODNAME
**REFCLASS** is the name of the reference class to use, and **TESTCLASS** is the name of the test class to use. Both of these must be in the classpath. **METHODNAME** is the name of the method to be tested.

The program will either output “Successful Run” on success, or a list of failures if any outputs don’t match between the reference and test classes. The output can also be tweaked to be more verbose.

### 4.4 Web-IDE Evaluator

JSymTester was also adapted to be used as a WebIDE evaluator. WebIDE evaluators are web services which can be used to evaluate some piece of student code. Specifically, the JSymTester evaluator takes three arguments: **refClass**, **testClass**, and **methodName**. These mirror the parameters of the command line version, but the classes are specified in source code form rather than as a class on the classpath. The evaluator compiles these on-the-fly and runs them through the JSymTester, returning the same results as the command-line version.

To make use of this evaluator easier, there is also a version which takes both pieces of code in the form of method definitions, rather than the code for whole classes. This allows instructors writing labs for WebIDE to have the students just write a simple method without having to wrap it in a class. This evaluator encapsulates the source code in a class automatically, then compiles it and sends it to the JSymTester just as the first evaluator does.
Chapter 5

Validation

The most important aspect of the JSymTester is that it is able to identify whether student code behavior matches the reference implementation. To this end, this section of this thesis attempts to compare the results of the JSymTester to the results from the more traditional unit tests or scripts written to test student code.

5.1 Test Setup

The first step in performing this comparison was to determine where to get sample student inputs. In order to do this in a sensible way, I chose to pass all student inputs for two classes for two Web-IDE labs to the JSymTester. I modified two previously-written introductory labs focusing on if statements and functions, adding the JSymTester evaluator to all of the evaluation steps which used unit testing or Java function call output comparison. This passed student code, along with a reference implementation of the exercise, to a JSymTester evaluator.
This modified JSymTester evaluator always returned success to the user, in order to avoid allowing any problems in its methods or implementation to affect student progress. However, it allowed me to collect a number of code samples along with the results of the previously-written evaluators. I then compared the results of the old evaluators with that of the JSymTester.

The hope was that the JSymTester would not report any submissions as correct for which the manually-written tests would fail (unless those tests happened to be written poorly). In addition, it could catch errors in student code which the manually-written tests do not; this would be further evidence of its efficacy.

5.2 Results

The 11 exercises across these two labs received a total of 1306 submissions from students across two classes. Of these, 806 submissions were able to be compiled and run without timing out. One of these submissions used some math which was not solvable by the Choco constraint solver used by JSymTester, so I have excluded it from the results (it was not implemented correctly, in any case).

I recorded all of the results for the remaining 805, both from the JSymTester and from the original evaluators. I also manually inspected all submissions which did not textually match a known good solution, in order to verify correctness. The results can be seen in Table 5.1. A “Success” represents a piece of code which passed the given validation, while a “Failure” means it failed that validation.

<table>
<thead>
<tr>
<th></th>
<th>Original Evaluators</th>
<th>JSymTester</th>
<th>Manual Inspection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successes</td>
<td>554</td>
<td>570</td>
<td>519</td>
</tr>
<tr>
<td>Failures</td>
<td>251</td>
<td>235</td>
<td>286</td>
</tr>
</tbody>
</table>

Table 5.1: Overall test results
The results show that the JSymTester is slightly behind the original evaluators in overall accuracy. It does not catch all of the problems that were caught by the original evaluators. However, if we inspect the data at the level of the individual exercises, the situation changes slightly. Table 5.2 shows the results for each exercise, with Success shortened to “S”, and Failure shortened to “F.”

<table>
<thead>
<tr>
<th>Exercise</th>
<th>JST S</th>
<th>JST F</th>
<th>Orig. S</th>
<th>Orig. F</th>
<th>Actual S</th>
<th>Actual F</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>43</td>
<td>0</td>
<td>41</td>
<td>2</td>
<td>41</td>
<td>2</td>
</tr>
<tr>
<td>AA2</td>
<td>42</td>
<td>40</td>
<td>50</td>
<td>32</td>
<td>38</td>
<td>44</td>
</tr>
<tr>
<td>AC</td>
<td>99</td>
<td>5</td>
<td>54</td>
<td>50</td>
<td>54</td>
<td>50</td>
</tr>
<tr>
<td>C</td>
<td>36</td>
<td>7</td>
<td>36</td>
<td>7</td>
<td>36</td>
<td>7</td>
</tr>
<tr>
<td>D</td>
<td>43</td>
<td>2</td>
<td>43</td>
<td>2</td>
<td>43</td>
<td>2</td>
</tr>
<tr>
<td>B.</td>
<td>69</td>
<td>22</td>
<td>80</td>
<td>11</td>
<td>69</td>
<td>22</td>
</tr>
<tr>
<td>C.</td>
<td>49</td>
<td>39</td>
<td>56</td>
<td>32</td>
<td>49</td>
<td>39</td>
</tr>
<tr>
<td>D.</td>
<td>61</td>
<td>54</td>
<td>64</td>
<td>51</td>
<td>61</td>
<td>54</td>
</tr>
<tr>
<td>E.</td>
<td>36</td>
<td>48</td>
<td>38</td>
<td>46</td>
<td>36</td>
<td>48</td>
</tr>
</tbody>
</table>

Table 5.2: Test Results By Exercise.

Table 5.2 shows that nearly all of the cases where JSymTester failed were in one particular exercise. This is even easier to see in Table 5.3, which shows the number of failures not caught by each evaluator (excluding those exercises where the evaluators missed nothing).
Table 5.3: Failures Not Caught By Evaluators.

<table>
<thead>
<tr>
<th>Exercise</th>
<th>Caught only by JST</th>
<th>Caught only by Orig.</th>
<th>Caught by neither</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>AA2</td>
<td>8</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>AC</td>
<td>0</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>31</td>
<td>47</td>
<td>4</td>
</tr>
</tbody>
</table>

The troublesome exercise, which I have labeled AC, gave the student the task of implementing an age calculator. This method took the current year, month, and day, and well as the year, month, and day of a person’s birthday. It then returned the age of this person in years, rounding down, as is commonly done.

Most of the student implementations for this exercise did not use any branches at all. Neither did the reference implementation. This meant that after one pass through the method, there were no constraints gathered to be falsified. The concolic execution would cease, providing only one input—usually 0 for every single argument. This meant that only this one input was being passed to student ageCalculator implementations. Many of these implementations were incorrect, but happened to give the correct results for this case.

This reveals a larger problem with JSymTester and concolic testing in general: They are aimed at generating inputs to maximize code coverage. They do not generate inputs to test particular mathematical equations. This is the cause of the mistakes in exercise AC as well as the six other misses in exercises AA and AA2.
This is a deficiency in concolic testing, so it is slightly out of the scope of this paper. Some possible solutions are provided in Chapter 6, however.

Dropping the AC exercise from the analysis would put JSymTester in a much more favorable position, mistaking only six bad implementations for good ones, and correctly identifying 31 incorrect implementations.

This provides evidence that the JSymTester, while it underperforms on code which has few branch points, actually performs as well or better than the traditional approaches on code which has more branch points.
Chapter 6

Future Work

There are a number of options for expanding the potential uses of the JSymTester. In addition, there are also many possible avenues of research related to the use of automated test generation in automated student code assessment.

6.1 Beyond Code Coverage

The key problem explained in the Results section of this thesis is that the JSymTester is designed to find inputs which provide good code coverage, not to find all possible interesting inputs. This means that for many programs which have few branches, the JSymTester will find only a small number of inputs. These inputs may not fully test the code.

For example, consider a function which simply takes an integer \( x \) and returns \( x + 2 \), and another function which takes \( x \) and then just returns 2, ignoring \( x \) completely. JSymTester would likely not find any differences between these two functions. This is because the JPF will start out by giving each function
an \( x \) value of 0. In each case, since there are no branches, it will gather no path constraints, have nothing to falsify, and consider its input generation done. Passing an \( x \) value of 0 to both of these functions results in a return value of 2, passing JSymTester’s comparison of the output of the two functions.

Obviously, this can lead to problems for certain coding exercises. There are a couple solutions to this which could be implemented in the future.

First, some extra values could be picked to run through the program. This could be done either only for the first run through the program, or for all. It may be worthwhile to pick a few random inputs to start with, or use a couple different inputs for each path constraint. This would require some extra work integrating with the constraint solver to force it to give multiple solutions to the same path constraint. Alternative test generation strategies, such as evolutionary testing, could also be applied here to generate these additional values.

Second, it may be possible to compare the symbolic return values of the methods in question. JSymTester already records these in terms of the symbolic input variables, so the data is available. This would also require some more integration with the constraint solver, however, since it would require comparing constraints for logical equivalence. Just comparing the particular representation of a set of constraints would not be accurate, as the constraints may have been gathered be in different orders or in different terms (for example, \( x < 5 \) rather than \( x \leq 4 \)).
6.2 Language Feature Support

The current implementation of the JSymTester focuses only on testing code at the method level. It supports inputs and outputs in the form of all Java primitives, as well as any objects composed only of these primitives. This does not include many of the classes provided by the Java class library, including Strings.

As it is currently written, JPF’s symbolic execution can support either lazily-initialized objects (as it used by JSymTester), or Java String objects, but not both at the same time. JSymTester could likely bypass some of this by running the JPF twice, once in each mode. The added complexity of this was not implemented for this thesis, as none of the student code examined used Strings in branch conditions.

It would also be useful to support more options of units to test. JSymTester itself supports testing only methods. Adding testing of constructors would be helpful for evaluating student object construction. However, the JPF’s symbolic execution framework is currently targetted at testing methods, meaning that it may be somewhat difficult to make these changes without also modifying the symbolic execution module.

JSymTester could also add some options for testing out class implementation as a whole, looking at calling different methods of objects in different orders. This would require taking advantage of other topics in test generation (namely, method sequence exploration).[13]

In addition, the way which JSymTester reconstructs objects means that it does not support comparison by reference of the input objects (as all input objects are created separately). This should be possible to rectify with a more advanced
integration with the JPF, as it handles this case.

6.3 Ease of Use

There are a few things which could be added to the tool to make it easier to use. The first of these would be removing the need for a default constructor to exist for any objects used as parameters to the methods being tested. This is a limitation of the Jackson ObjectMapper being used to reconstruct the objects, but should be easily overcome using bytecode manipulation (much as was done for the required main() method).

In addition, it would be useful to allow the instructor to specify an equals method to be used to compare the output of the test code with that of the reference code. As it is, the system uses the equals method of the object being returned. A custom method would mean that instructors could compare only a subset of fields, or use different comparisons for different exercises without having to modify the classes in use.

6.4 Model-Checking

It is worthwhile to note that JSymTester does not take advantage of any of the model-checking capabilities of the JPF. This may be a useful addition in the future, to allow instructors to be able to write something which can more deeply inspect the student’s code as it runs to check for certain things other than just correct output. This would, however, require the instructor to learn a bit about the JPF, something which JSymTester does not require as currently implemented.
6.5 Constructive Feedback

Finally, it may be worthwhile to assess the effectiveness of the feedback which JSymTester (or any other such tool) can provide. With the path constraints created with symbolic execution, it is possible to give the student feedback which defines the conditions under which their code fails in the abstract rather than stating exact values.

For example, instead of saying “Your code failed for inputs: \( x = 5, y = 6 \),” the JSymTester could reply instead with a more informative message like “Your code failed under the following conditions: \( x < y, y > 5 \).” This allows students to approach their error from the standpoint of conditions which actually exist in their code, rather than focusing on specific inputs (in the latter case, they might even be tempted to simply add a specific special case for a failing input!).

To evaluate which of these approaches might be better, it would be best to try both approaches with different sets of students to see which group finds the messages more helpful.

Current research suggests that the abstract information may be more helpful.[13]
Chapter 7

Conclusion

The JSymTester is a new kind of student assessment tool utilizing symbolic execution to automatically generate tests for student code that are actually based on the conditions in that code. This is a benefit for professors, as it is easy to simply implement a reference implementation of a programming exercise, instead of having to write tests which can only hope to cover all of the edge cases which may be present in student code.

The JSymTester is based on the Java PathFinder symbolic execution module, and so can benefit from any additional capabilities added to it. It is available both as a command line tool, and as a Web-IDE evaluator.

In a simple trial comparing the JSymTester WebIDE evaluator against the more traditional output comparison evaluators, the JSymTester behaved similarly to the original evaluators. Though it behaved poorly in one case, in other cases, it found errors that the traditional testing did not. This provides evidence that the JSymTester can effectively supplement output comparison, and, with a few more tweaks, even replace it. Furthermore, this shows that automated test generation
techniques are applicable in the area of student code assessment.
Bibliography


