JMLCUTE: Automated JML-Based Unit Test Case Generation

Rafael Baltazar
Instituto Superior Tecnico, Lisboa, Portugal,
rafael.baltazar@tecnico.ulisboa.pt

Abstract. A formal specification is the detailed description of a function’s behaviours. Formal specification can be helpful during development by providing a formal documentation on the behaviour of complex functions. Some verification tools already support formal specifications to automatically detect program errors and automatically generate a useful error message or a representative test case. However, existing verification tools that automatically detect program errors and support formal specifications cannot extract information from unknown or complex functions and end up with an incomplete verification. This article proposes JMLCUTE: the first tool to automatically detect program errors, support a formal specification and use concolic testing - a method that uses both program and run-time analysis to deal with unknown or complex functions. This article also presents an evaluation on how formal specification and concolic testing use each other to generate new and interesting test cases. The evaluation compares JMLCUTE with a similar concolic testing tool that does not support formal specifications, jCUTE. Evaluation results show that JMLCUTE can find specification errors that jCUTE ignores, and that both tools’ execution times are similar for complex projects.

Keywords: automated test case generation, test-driven development, program verification, formal specification, symbolic execution, concolic testing

1 Introduction

As a software product increases in complexity, it becomes harder to assure its correct behavior. To assess a software product’s correctness, developers cannot rely on intuition alone; developers must rely on software testing.

There are several types of software testing, but this article focus on one: unit testing. Unit testing verifies the correct behavior of the smallest components of a software product by executing and verifying unit test cases. A unit test case executes a program path and verifies at each program point whether the software product’s run-time behavior respects the expected behavior.

Unit testing can be executed manually. In this case, a developer manually writes the input to a unit and verifies whether the unit’s output respects the unit’s expected behavior.
However, manual testing is labor intensive and corresponds to 50% of the Research and Development budget of most companies that develop software products [1]. Therefore, developing techniques to automatically test software is of utmost importance to several research branches [2–12].

Each developed software testing and verification technique has its strengths and weaknesses and is usually tailored for a specific development process. This article focuses on two development processes: Test-Driven Development [3] and Design by Contract [13].

Test-Driven Development is an agile development process that guides a unit’s implementation by first generating a few representative test cases. This way, Test-Driven Development manages to maintain a certain level of software quality, while simplifying the module’s implementation as much as possible.

While Test-Driven Development focuses on a module’s implementation, Design by Contract focuses on a module’s design. Design by Contract designs a module by formally specifying contracts. Contracts are obligation-benefit relationships between a module and the other modules that use it.

Design by Contract can be combined with Test-Driven Development for bonus benefits. Design by Contract encourages modular development, which is crucial for Test-Driven Development, while Test-Driven Development reuses the contracts as both guides and oracles for creating test cases.

Unfortunately, there are no automated verification tools that take advantage of the aforementioned benefits to detect complex errors on a Test-Driven Development. Existing tools either only detect simple errors, or do not take a module’s formal specification into account when generating test cases, or take too long to generate test cases, halting the module’s implementation.

This article proposes JMLCUTE: an automated test case generation tool that uses concolic testing to generate input data and a module’s contracts for assertion checking.

Section 2 discusses the research branch of software testing and verification and some of the automated tools that test and verify software. Section 3 describes the main goals and architecture of the proposed tool: JMLCUTE. Section 4 presents the evaluation made to JMLCUTE. Finally, Section 5 concludes this article.

2 Software Testing and Verification

Software testing is a set of verification techniques that measures the correctness of a program, in which most verification techniques are semi-automatic. In some cases, manually written test cases are automatically executed in a testing framework, for example, JUnit [3] or TestNG [14] for Java. In other semi-automatic techniques, the developer guides the verification by providing additional information other than the program.

The type of information provided depends on the verification technique, and can range from providing formally written specifications to theorems that represent loop invariants at run-time.
A specification document consists of all expected inputs, outputs and behaviors. A specification can be written informally, for example, in a natural language using Javadoc, written formally, in a language fit for both humans and computers, like Java Modeling Language (JML), or unwritten, which is not recommended.

2.1 Related work

Boogie [8] is a fully automatic static verification tool that supports formal specifications and aims to find all types of errors. However, Boogie sacrifices speed over finding all errors, making it unfit for unit testing.

LOOP [15] is an interactive verification tool that supports formal specifications and aims to find all types of errors. Unfortunately, LOOP sacrifices user effort time over finding all errors, making it unfit for unit testing.

ESC/Java [11] is a fast static verification tool that partially supports a formal specification to find simple and common errors. ESC/Java’s speed is fit for unit testing, but ESC/Java’s error detection capabilities are small and ESC/Java is unable of generating any test case.

Korat [5] is an automated test case generation tool that supports formal specifications and uses exhaustive testing to generate test cases. Unfortunately, exhaustive testing generates a great number of output that the developer needs to manually process. Additionally, Korat can only detect pointer errors.

CUTE [6, 16] is an automated test case generation tool that uses concolic testing: a technique specifically tailored for understanding complex programs. Because of this, CUTE can detect complex arithmetic and pointer errors. CUTE’s speed is fit for unit testing, but CUTE does not support formal specifications.

After comparing the five tools, the tool most fit for unit testing is CUTE for its speed and error detection capabilities.

Table 1. Tool performance given the factors in the first row. All but LOOP are automated verification tools.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Error type</th>
<th>Time</th>
<th>User effort</th>
<th>Output</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boogie</td>
<td>Complex data</td>
<td>Hours</td>
<td>specification</td>
<td>Messages</td>
<td>JML, Spec#</td>
</tr>
<tr>
<td>LOOP</td>
<td>Complex data</td>
<td>Hours</td>
<td>specification, user expertise</td>
<td>Messages</td>
<td>JML</td>
</tr>
<tr>
<td>ESC/Java</td>
<td>Simple errors</td>
<td>5 minutes</td>
<td>optional specification</td>
<td>Messages</td>
<td>JML</td>
</tr>
<tr>
<td>Korat</td>
<td>Pointer constraints</td>
<td>30 minutes</td>
<td>optional specification</td>
<td>Test cases</td>
<td>JML</td>
</tr>
<tr>
<td>CUTE</td>
<td>Integer, boolean or pointer</td>
<td>5 minutes</td>
<td>optional specification</td>
<td>Test cases</td>
<td>Tool-specific</td>
</tr>
</tbody>
</table>
3 JMLCUTE

JMLCUTE is a verification tool that automatically analyzes the program under test, the program’s specification and the program’s run-time behavior to automatically detect errors and generate a unit test case per error found.

Existing tools either detect only simple errors [11], lack specification support [6, 16], or cannot give feedback about errors within unit testing time [15]. JMLCUTE balances these three aspects optimizing them for unit testing in a Test-Driven Development.

A JMLCUTE execution consists of five steps (figure 1):

Parse and Typecheck JMLCUTE needs to compile both Java and JML and, like any other Java compiler, JMLCUTE parses and typechecks any of the Java project files annotated with JML. This step is already present in the AspectJML compiler and is not modified.

Instrument JMLCUTE instruments the compiled project files to detect errors of specification violation. This step is modified to insert calls to a concolic testing library.

Generate Entry Point Since the simplification module requires one entry point at a time, the entry point generator wraps the method under test in a main method. This step is custom made and does not exist in other tools.

Simplify Entry Point Java bytecode syntax is complex. Since JMLCUTE performs program analysis, it is best to translate the bytecode into a simpler version. This step already exists in jCUTE and is not modified.

Execute Concolic Tester Once the entry point is simplified and the concolic tester library calls have been added, the program can be executed several times. Each execution further explores the branches of the program and, if a new branch was executed, generates a test case. This step is modified to generate test cases with explanatory comments.

4 Evaluation

One of JMLCUTE’s objectives is to generate new and interesting test cases that were previously ignored by tools who did not support formal specification. This evaluation attempts to understand how effectively do formal specifications and concolic testing use each other to generate new and interesting test cases. To do that, we compare JMLCUTE’s effectiveness to that of jCUTE’s.

However, jCUTE is a concolic testing engine - not a complete tool - and does not support efficient evaluation of multiple methods. To simulate jCUTE, we disable our AspectJML compiler and use a regular Java compiler, javac, instead.

4.1 Metrics

The first metric to capture during the evaluation is the execution time of each step - AspectJML or javac compilation, simplification, and concolic testing -. Since JMLCUTE adds instrumentation code to the program before concolic
testing, JMLCUTE is expected to execute slower than jCUTE. However, it is only acceptable that JMLCUTE uses significant more time than jCUTE, if JMLCUTE increases the number of executed paths or branches.

The second metric is the number and percentage of covered branches. The greater the percentage of covered branches, the higher the efficacy of the tool.

The final metric is the number of errors found. Since JMLCUTE adds specification code to the bytecode before the concolic testing, the number of errors found is expected to increase, when comparing to jCUTE.

4.2 Setup

We run the evaluation on a Windows 7 Home Premium 64 bits, with an Intel(R) Core(TM) i3 Central Processing Unit with two cores of 2.27 GHz each, and 3.85 GB of usable Random Access Memory.

Since JMLCUTE only runs on a Linux 64-bit system, we use Vagrant version 1.7.4 with Oracle VM VirtualBox version 5.0.2 to run a virtual machine. The virtual machine is an Ubuntu 64-bit system, with access to 2048 MB of RAM, and a processor execution cap of 50%.

Java runs with default arguments for compiling and instrumenting JML, and `-Xmx2000m` for concolic instrumentation and execution. The Java version used is 1.7.0_65 java(TM) SE Runtime Environment and the java compiler used is javac version 1.7.0_65.

Both JMLCUTE and jCUTE run with the argument `-r` to randomly generate input integer values, instead of using the default integer value, zero.
4.3 Benchmarks

We used two benchmarks: the Samples benchmark, custom made and consists of classes, each one demonstrating one type of constraints, and the Event-B benchmark, edited version of a benchmark available at the EventB2Java Rodin plug-in website.

The Event-B benchmark consists of 109 classes divided into nine projects. Each project consists of events (functionality) that alter a given machine (data). Each project also uses the custom collection library `eventb_prelude` that implements several Event-B objects.

The benchmark is edited to suit AspectJML and jCUTE limitations. Namely, lack of Java 5 features - generics in the interface or specification, auto-boxing - , errors in the JML semantics implementation - checking for static invariants before static initialization is complete, name collision when dealing with specification quantifiers - , and jCUTE limitations - adding constructors with no arguments. Therefore, we removed all generic references, added auto-boxing, removed some of the specification invariants, history constraints.

4.4 Samples Evaluation Results

JMLCUTE takes almost twice as much execution time as jCUTE to concolically test the Samples benchmark. This result is expected, because the AspectJML compiler is several times slower than the javac compiler, and the specification of the Samples benchmark has more lines of code than all methods' implementations combined.

The evaluation results for branch coverage (table 2) can be misleading. While JMLCUTE has a lower branch coverage percentage than jCUTE, some branches in the concolic tester generated by JMLCUTE are not supposed to be covered. These uncoverable branches belong to AspectJ-specific methods that check for advice validity at run-time.

While jCUTE fails to automatically detect any error, even when the test case triggers the error, JMLCUTE finds all specification errors. This shows that even when JMLCUTE does not achieve 100% branch coverage in the entire program, JMLCUTE can achieve 100% branch coverage on the meaningful branches, i.e. the branches that lead to the discovery of errors.

4.5 Event-B Evaluation Results

The evaluation of the smaller projects of the Event-B benchmark (figure 2) shows that JMLCUTE takes up to twice as long as jCUTE to concolically test the entire project. Most of the execution time is spent simplifying the concolic tester, so reducing the complexity of the inserted code by the AspectJML compiler should reduce this gap.

Table 2. The branch information gathered from the evaluation of the Samples benchmark. Each character in the first row represents one of the evaluated methods:
A: `ConjunctionOnCollection.guard(BSet,Integer)`,
B: `ConjunctionOnCollection.isSubsetGuard(BSet,Integer)`,
C: `ConjunctionOnCollection.guardCustomBSet(CustomBSet,Integer)`,
D: `ConjunctionOnCollection.guardPublicCustomBSet(PublicCustomBSet.Integer)`,
E: `ConjunctionOnIndirectCollection.guard(BSet,Integer)`,
F: `LinearArithmetic.add(int,int)`,
G: `LinearArithmetic.has(int)`,
H: `NonNull.guard(Object)`,
I: `NonNull.complexGuard(BSet)`.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total branches</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmlcute</td>
<td>20</td>
<td>34</td>
<td>14</td>
<td>14</td>
<td>18</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>jCUTE</td>
<td>8</td>
<td>22</td>
<td>6</td>
<td>22</td>
<td>16</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Branches covered</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmlcute</td>
<td>10</td>
<td>7</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>8</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>jCUTE</td>
<td>6</td>
<td>9</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Branches covered (%)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmlcute</td>
<td>50</td>
<td>38.2</td>
<td>75</td>
<td>42.9</td>
<td>22.2</td>
<td>80</td>
<td>83.3</td>
<td>78.6</td>
<td></td>
</tr>
<tr>
<td>jCUTE</td>
<td>75</td>
<td>40.9</td>
<td>50</td>
<td>100</td>
<td>31.3</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

When comparing the execution times of both JMLCUTE and jCUTE to the total number of branches of each project (figure 3), we see that JMLCUTE outperforms jCUTE in complex projects. This is an interesting result, because both JMLCUTE and jCUTE differ only on the complexity added by the AspectJML compiler, so the execution times should be the same, when the program complexity is the same.

This phenomena occurs when a test case ends abruptly, before the method is executed. For example, some classes in the MIO_multi_threaded project may violate the specification, when the constructor for the class is called. This renders the test case useless - because the method under test is never the constructor -, but also finishes the concolic testing of that class sooner. In one way, for the same program complexity, JMLCUTE fails faster than jCUTE, which leads to a decreased execution time.

Once again, JMLCUTE sees an expected decrease in the percentage of branches covered, when comparing JMLCUTE with jCUTE. However, the number of branches covered doubled, when comparing JMLCUTE with jCUTE, hinting that a powerful enough constraint solver could solve all added constraints.

All Event-B methods contain mostly collection constraints, i.e. constraints the use classes from the Java Collection framework. Since neither JMLCUTE nor jCUTE are capable of solving collection constraints, neither JMLCUTE nor jCUTE found any errors.

4.6 Evaluation Summary

JMLCUTE was capable of concolically testing 109 classes in approximately the same execution time as jCUTE, outperforming jCUTE on complex projects.
Fig. 2. The execution times of both jCUTE and JMLCUTE for each project of the Event-B benchmark. The times measured were compilation time - time spent in the AspectJML compiler for JMLCUTE, or javac for jCUTE -, instrumentation time - time spent in the simplification component -, and generation time - which includes concolic testing and test case writing.

Fig. 3. Execution times related to the total number of branches of each project for both JMLCUTE and jCUTE. The number of branch total instances is greater than the number of projects, because JMLCUTE and jCUTE generate a different number of branch totals for each project. Note that the horizontal axis is not scaled linearly.
The Event-B benchmark presented a challenge to jCUTE’s constraint solver. Since jCUTE’s constraint solver cannot solve collection constraints, neither JMLCUTE nor jCUTE could solve most of the constraints of the Event-B benchmark.

However, the evaluation results of the Samples benchmark show that both JMLCUTE and jCUTE are capable of solving linear arithmetic, boolean and simple pointer constraints. Additionally, JMLCUTE outperformed jCUTE by finding specification errors, which jCUTE ignored.

5 Conclusion

This article proposes JMLCUTE, the combination of formal specification with concolic testing tailored for projects that follow Test-Driven Development and Design by Contract.

JMLCUTE improves an already existing concolic testing engine, jCUTE, by instrumenting the program’s bytecode with JML assertion checks. JMLCUTE also improves jCUTE by generating entry points for each method of each class under test, such that concolic testing on multiple classes becomes fully automatic.

Evaluation on both JMLCUTE and jCUTE shows that JMLCUTE can find specification errors that are otherwise ignored by jCUTE. The execution time of both JMLCUTE and jCUTE are similar; JMLCUTE is slower than jCUTE on simple projects, but JMLCUTE catches up on bigger projects, becoming even faster than jCUTE in one project.

5.1 Future Work

JMLCUTE can be further improved in two areas. The most important change is improving JMLCUTE’s constraint solver to handle collection constraints, because real-world software projects use the Java Collection framework extensively, directly or indirectly. A possible alternative is the Choco 3 [17] constraint solver that is capable of solving graph constraints.

On small projects, most of the execution time of JMLCUTE is spent on simplifying and instrumenting the project’s bytecode. The simplification time of JMLCUTE is approximately twice the simplification time of jCUTE. JMLCUTE’s simplification time can be reduced by either reducing the size of the inserted code or optimizing the simplification component.

References


