JMLCUTE

Automated JML-Based Unit Test Case Generation

Rafael Farias Baltazar

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. João Carlos Serrenho Dias Pereira
Prof. Pedro Miguel dos Santos Alves Madeira Adão

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. João Carlos Serrenho Dias Pereira
Member of the Committee: Prof. Luís Jorge Brás Monteiro Guerra e Silva

November 2015

Acknowledgments

I would like to thank my advisors who helped me schedule, organize and edit this dissertation, and the examination committee who evaluated and provided feedback for the final version of this dissertation. Finally, I would like to thank my friends and family who provided feedback and supported me throughout this past year.

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 dissertation 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 dissertation 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

Resumo

Uma especificação formal é uma descrição detalhada do comportamento de um método. Especificações formais podem auxiliar o desenvolvimento de software servindo de documentação para métodos complexos. Algumas ferramentas de verificação de software suportam especificações formais para detetar errors num programa e gerar automaticamente uma mensagem de erro ou um caso de teste representativo.

No entanto, ferramentas de verificação existentes que detetam erros automaticamente e suportam especificações formais não conseguem extrair informação de métodos complexos ou cuja implementação é desconhecida, e terminam com uma verificação incompleta.

Esta dissertação propõe JMLCUTE: a primeira ferramenta que deteta errors num programa automaticamente, suporta especificações formais e usa verificação "concólica" - uma técnica que analisa a execução do programa para lidar com métodos complexos ou desconhecidos.

Esta dissertação também apresenta uma avaliação que testa o quanto especificações formais e a verificação "concólica" se auxiliam para gerar casos de teste novos e interessantes. A avaliação compara JMLCUTE com uma ferramenta de verificação "concólica" que não suporta especificações formais, jCUTE. Os resultados da avaliação mostram que JMLCUTE encontra erros de especificação que jCUTE ignora, e que os tempos de execução de ambas as ferramentas é semelhante para projectos complexos.

Keywords: geração automática de casos de teste, desenvolvimento orientado a teste, verificação de programas, especificação formal, execução simbólica, execução concólica

Contents

List of Tables

List of Figures

Listings

Acronyms

AST Abstract Syntactic Tree. 17, 18, 29

CPU Central Processing Unit. 35

JML Java Modeling Language. 15, 16, 22, 23, 25–27, 41, 65, 68, 69

JVM Java Virtual Machine. 33

RAC Run-time Assertion Checking. 16–18, 27, 34

RAM Random Access Memory. 35

Chapter 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 dissertation is focused 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, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12].

Unit test cases have three steps that can be automated: input data generation, execution and assertion checking (oracles).

Input Data Generation Input data generation establishes the state of the program before executing the method under test. Techniques that automatically generate data include random generation [4] and static verification [13].

Execution Once the input data is generated, the test case can be executed. There are several unit testing frameworks [14, 15] that automatically execute unit test cases.

The unit testing frameworks' main quality is the reusability of the written test cases. Write the test case once, and execute it throughout the development.

Assertion Checking To decide whether a test case fails or passes, the developer needs to understand the program's expected behavior, i.e. the program's specification.

A program's specification can be written formally, using a formal specification language that can be read by both developers and computers, written informally, using natural language, or simply unwritten.

Once a developer knows the the program's specification, the developer can use assertion checks that stop the test case whenever the program's specification is violated.

By attempting to automate the above steps, researchers developed several software testing and verification techniques. Techniques like full static verification, fast static verification, and concolic testing.

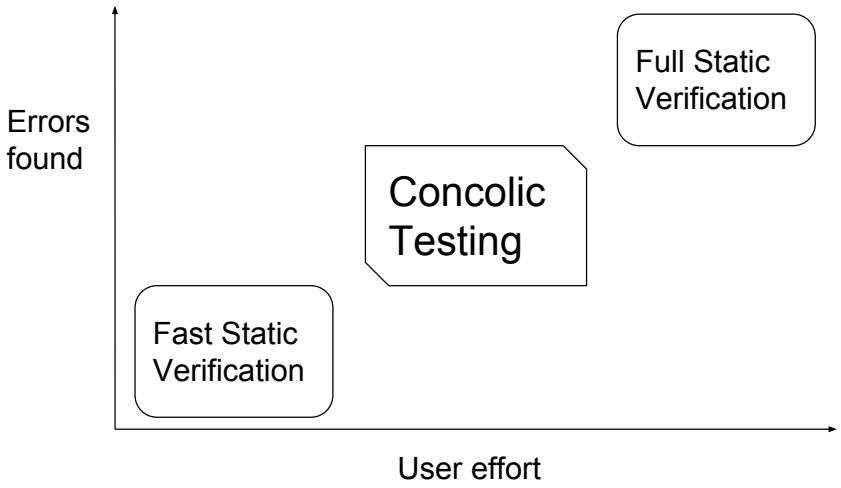

Figure 1.1: Comparison of automated software testing and verification techniques ordered by user effort and number of errors found.

Full static verification aims to find all errors in a program without executing the program. Full static verification is known for its slow but highly effective error detection.

Fast static verification aims to find as many errors as possible in a given time limit also without executing the program. Therefore, fast static verification can be used to find common or simple errors in any stage of the development.

A third verification technique, concolic testing, stands between full and fast static verification (figure 1.1). Concolic testing analyzes and executes the program under test to detect complex errors. Concolic testing can only detect errors during program execution. Therefore, whenever concolic testing finds an error, the input data that triggered the error is already available, which makes concolic testing perfect for input data generation.

Each software testing and verification technique has its strengths and weaknesses and is usually tailored for a specific development process. This dissertation is focused on two development processes: Test-Driven Development [3] and Design by Contract [16].

Test-Driven Development is an agile development process that guides a unit's implementation by first generating a few representative test case. 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 is focused on a module's implementation, Design by Contract is focused 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 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 [5, 4, 11, 12], or do not take a module's formal specification into account when generating test cases [6, 17, 10], or take too long to generate test cases [18, 8], halting the module's implementation.

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

1.1 Objectives

JMLCUTE is tailored specifically for a Test-Driven Development that uses Design by Contract, such that the developer effort put into writing a module's contracts is automatically reused to generate that module's test cases.

JMLCUTE aims to:

- detect complex errors in early stages of module development,
- decrease developer effort in writing a test suite.

1.2 Outline

Chapter 2 discusses the research branch of software testing and verification and some of the automated techniques that test and verify software. Chapter 3 describes some of the state-of-art tools on automated software testing and verification. Chapter 4 describes the main goals and architecture of the proposed tool: JMLCUTE. Chapter 5 extensively describes the goals, origin and modifications made on JMLCUTE's components. Chapter 6 presents the evaluation made to JMLCUTE. Finally, Chapter 7 concludes this dissertation.

Chapter 2

Software Testing and Verification

There are several techniques to assure software quality during development, for example, following code styles, making code reviews and testing software.

Software testing is a set of verification techniques that measure the correctness of a program. Software testing can be manual, automatic or semi-automatic.

In manual software testing, a human executes the program and manually inputs the test case. Later, the human analyzes the program's output and manually calculates quality metrics.

In automatic software testing, a tool can either create a set of test cases to discover errors in the program - black-box testing -, or analyze the program and infer errors or warnings from the analyzed program - white-box testing.

However, most verification techniques are semi-automatic. In some cases, manually written test cases are automatically executed in a testing framework, for example, JUnit [3] 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. Semi-automatic verification techniques will be further discussed below.

2.1 Specification

All expected inputs, outputs and behaviors of a program form the specification document.

The specification can be written informally, written formally or unwritten.

Informal specification The informal specification follows no particular pattern or template and is focused on human readability. Informal specification can be written in english or other natural language. Javadoc is an example of a tool for java that supports informal specifications. Javadoc uses natural language java comments on methods, classes and packages as both specification and documentation.

Formal specification Like any specification, the formally written specification also describes a program's behavior, but is focused on both human and machine readability. Formal specifications are usually more expensive to write than informal specifications, but, given their formal syntax and semantics, formal specifications can be used by automated tools for various goals.

Unwritten specification Every software product has a specification, whether the project team writes it or not. Not writing a specification may have consequences later in development, when the team

no longer remembers the expected behavior of existing modules. Reading the source code of the modules may prove useful only if the module is correctly implemented. Nonetheless, specification languages are usually created with the intent to summarize a module's behavior.

2.2 Development Processes

Maintaining software quality throughout development through software testing and verification supports further improvements by providing a reliable base of knowledge about the program. However, software testing carries a deadly cost: developer and computer effort. To adapt to this cost, each verification technique is made specifically for a certain type of development process.

This paper will not address all development processes and will focus on Test-Driven Development [3] and Design by Contract [16].

2.2.1 Test-Driven Development

Test-driven development is an agile development process designed for smaller teams and is based on a very short development cycle.

1. The developer writes a few test cases for a module;
2. The developer codes the module, until it passes all test cases;
3. The developer refactors the code according to given standards;
4. The developer writes a few more test cases and the cycle repeats.

This short cycle helps maintain software quality throughout the development by helping the team detect errors early. Given the fast-paced environment of Test-Driven Development, fast verification techniques that both systematically detect errors and provide useful messages on how to fix errors are specially useful.

2.2.2 Design by Contract

While the Test-Driven Development is focused on software implementation, Design by Contract is focused on software design.

In Design by Contract, the developer formally specifies modules with contracts. Contracts establish a (business) relationship of obligation-benefit between modules (figure 2.1); if a module uses another module, the used module must behave according to used module's obligation contract - postcondition -, but only if the input respects the used module's benefit contract - precondition.

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. Some verification techniques can also use this development process setup to automatically generate test cases from the contracts and provide an initial test suite.

2.3 Challenges

Type of errors found The type of errors found by a verification technique depends on both the error and the program's complexity. Some errors are hidden in a way that makes them impossible to find

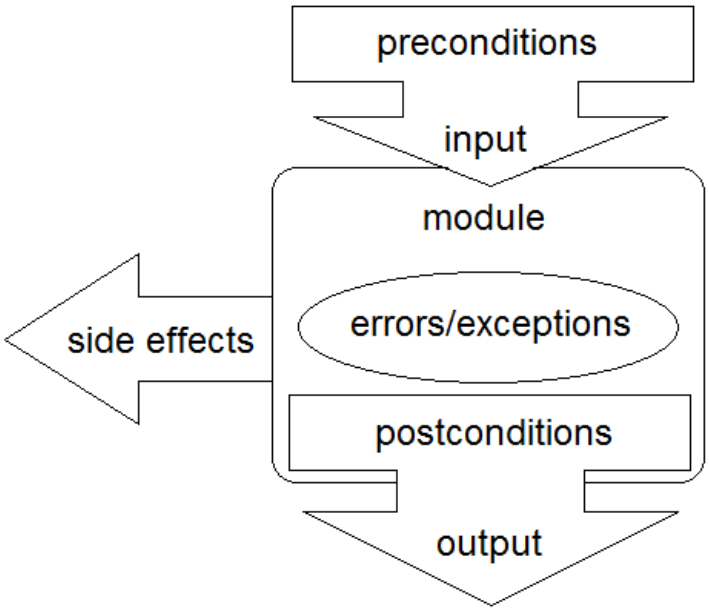

Figure 2.1: Sample contract for a software module composed of preconditions and postcondition. It is the module's responsibility to only output values that respect the postcondition. However, if the module is used by another module, it is the other module's responsibility to only input values that respect the precondition. This way, the module only needs to behave correctly in a smaller set of the input and the other module can expect the behavior described in the contract.

with a general technique - undecidability problem. Other errors - specification violations - can only be found in the presence of readable, precise specification. Finally, programs may be so complex by design (encryption functions) that it would take years to analyze all possible program paths.

Computer time If a verification technique aims to find all errors automatically, it will have to face the exponential nature of program complexity. For every new branch in a program, the number of possible program paths can double - for if statements - or increase theoretically infinitely - for loop statements. A technique that systematically explores all program paths may never end. Verification techniques can then implement a timeout to their analysis or depend on developer effort to finish their analysis.

Developer time Verification techniques may request developer effort to complete program analysis. Developers are much smarter than machines, even if more error-prone, and can usually prove whether the program respects a certain contract through logical jumps that machines are incapable of. If the verification technique fails, developers can modify the program or the specification to guide However, developer time may be the most powerful resource, but it is also the most expensive.

Output utility Verification techniques have several ways to reduce the amount of time needed to find program errors; many of them passing through simplifying the program. Simplifying the program may reduce program size and complexity, but usually implies a loss of information about errors. The information loss makes it difficult to print useful error messages or generate test cases to later fix the error. To increase the utility of the output, verification techniques require a second form of program transformation to add line numbers, file names, among other useful information, when printing error messages or generating test cases.

There is no verification technique that is capable of finding all types of errors on any program, in zero

time, fully automatically, and provides useful output. The value of a verification technique is in how well the trade-off between benefits applies to real software development processes.

2.4 Verification Techniques

Several techniques aim towards the previously mentioned goal: find program errors and provide feedback on how to fix them.

Of the many verification techniques used in software verification, this section mentions three: static verification, symbolic execution and concolic execution.

2.4.1 Static Verification

Static verification is a technique that finds errors and gives feedback to the developer, usually in the form of error messages, without executing the program.

An example of such tools is embedded in javac: the Java Programming Language Compiler. javac uses typechecking (a form of static checking) to detect whether an illegal cast can be performed in the program. If javac detects an illegal cast, javac will print an error message with information on how to fix the error.

Static verification tools split into three groups:

Automated Full Verification An automated full verification tool tries to prove the absence of errors in the entire program. Whenever, the tool fails to prove the absence of an error, the tool prints an error message. These tools favor printing possible false errors over missing an error. An example of an automated full static verification tool is Boogie [8].

Interactive Static Verification Like automated full verification tools, interactive static verification tools are meant for system testing. Interactive static verification tools too aim to find all errors in a program, but do so in an interactive way; whenever the tool fails to prove the absence of an error, instead of writing an error message, the tool lets the user provide any missing information that may help complete the verification. An example of such tools is the LOOP tool [18].

Fast Static Verification In the opposite end of the spectrum, fast static verification aims to find interesting bugs in useful time.

Fast static verification's definition of useful time depends on where fast static verification is applied. If applied in unit testing, the verification has approximately 15 minutes to give feedback about the program. Additionally, if the static verification tool is fast enough, the tool may be integrated with an Integrated Development Environment to provide feedback on program errors as the developer is typing.

Fast static verification relies on speed. To achieve this, fast static verification may only partly analyze the program and specification, to provide feedback to the developer as fast as possible. An example of fast static verification tools is ESC/Java [11].

2.4.2 Symbolic execution

A technique a step further from static verification is automated test case generation. Automated test case generation, like static verification, aims to find errors in a program, but gives feedback in the form of test cases. The output test cases can be analyzed and debugged by the developer to more easily understand the error.

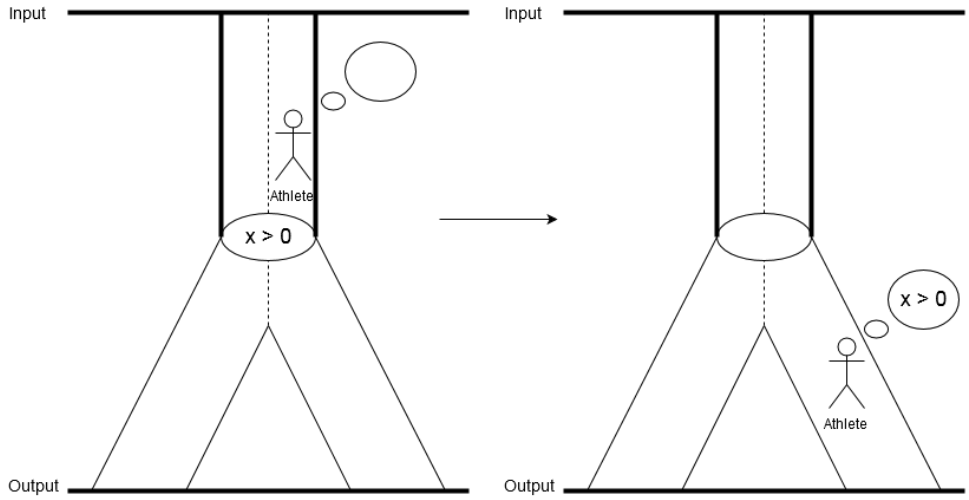

Figure 2.2: The athlete starts the race with no branches. When passing by the positive branch (represented by the ellipse $x > 0$), the athlete collects the branch ($x > 0$).

Symbolic execution is a technique that takes from both static verification and automated test case generation. Symbolic execution's goal is to generate a test case for each of a program's feasible paths, by statically analyzing the program. Most implementations of symbolic execution are capable of detecting errors and signaling which test cases lead to faulty executions.

A good way to understand how symbolic execution works is through a racing metaphor (figure 2.2). In the race, the athlete runs along a program path starting on the input and ending on the output. As the athlete moves, the athlete collects several branches. In the end, the athlete no longer remembers the input and must rediscover the input by using the collected branches and a deciphering technique.

Technically, symbolic execution has two phases: syntactic evaluation and constraint solving. The syntactic evaluation is represented by the branch collection, while the constraint solving is the deciphering technique.

Constraint solving is a general technique that generates values for a set of variables that satisfy a set of constraints. In symbolic execution, the constraints are the collected branches and the generated values are the test case.

Naively, symbolic execution seems to be able to detect any error and generate a test case for each error found. However, symbolic execution has several limitations:

Environment If the method under analysis is complex, the syntactic evaluation may generate an explosive number of paths (at least 2^n, where n is the number of conditional branches) which raises the execution time by an arbitrarily large amount. On the other hand, if the method's source code is unavailable (bytecode library, system call, etc.), it is impossible to collect the syntactic branches. These problems are known as the *complex or unknown environment* problem.

Non-determinism If the program is non-deterministic, the test case may execute a different path each time and therefore miss the error. This problem affects all automated test case generation techniques.

Undecidability Constraint solving is a powerful, but complex technique. Some types of constraints form undecidable theories, i.e., it may be impossible to know if there are values that satisfy the constraints. Solving non-linear arithmetic over a set of input variables is an example of an undecidable theory.

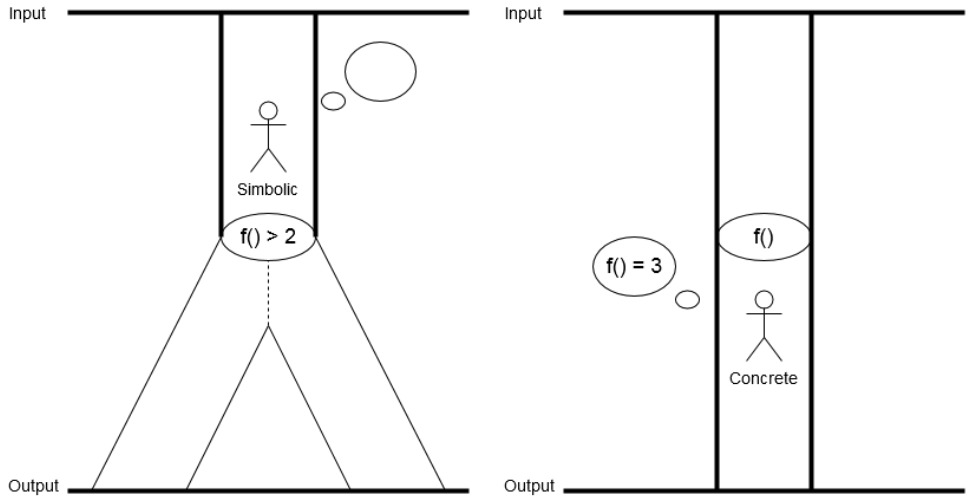

Figure 2.3: When the symbolic athlete finds a complex or unknown function, the concrete athlete runs through the program, until the concrete athlete captures the return value of the complex or unknown function.

2.4.3 Concolic Execution

The authors of DART [7] created concolic execution [7, 6, 17, 13, 10] to fix some of the problems of symbolic execution. Although the term “concolic” was created by the authors of CUTE [6, 17] as a mix of the terms symbolic and concrete.

Concolic execution has the same goal as symbolic execution, which is to generate a test case for each of a program’s feasible paths. Concolic execution sets itself apart from symbolic execution by using the return value of complex or unknown functions to solve part of the environment problem. The return value of a complex or unknown function is only available at runtime, so the concolic execution must capture the return value by executing the program.

Comparing with the symbolic execution’s race metaphor, concolic execution has two athletes: one symbolic and one concrete (figure 2.3). The symbolic athlete behaves the same way as the symbolic execution’s athlete; collecting branches and deciphering the test case. However, when the symbolic athlete finds a complex or unknown function, the concrete athlete runs the program and captures that complex or unknown function’s return value. This way, concolic execution simplifies the complex or unknown function by replacing its syntactic expression by its return value.

Concolic execution does not solve the environment problem completely. To capture the return value of a complex or unknown function, that function first needs input. Sometimes it is impossible to manipulate the complex or unknown function’s return value in order to assure that all branches are analyzed; concolic execution only assures one return value per complex or unknown function.

Chapter 3

Related Work

This section describes tools that implement one of the above techniques: static verification, symbolic execution or concolic execution. Each section will discuss the goal, architecture, strengths, and weaknesses of each tool.

3.1 Boogie Tool

Boogie [8] is an automated full static verification tool. Thus, Boogie automatically finds errors in a program and prints an error message for every possible error.

Every verification tool has its own definition of error and Boogie is no different. Boogie already detects a set of predefined errors, like dereferencing null pointers, but the developer can extend this set by using a specification language.

A specification language expresses program properties that must be true at certain points of the program. A program property violation is equivalent to an error, thus proving that an error exists is equivalent to proving that a program property is false at certain inputs. Boogie supports the specification languages Java Modelling Language for Java programs and Spec# for C# programs.

Given a program and its specification, Boogie finds errors by:

1. translating the program and its specification into theorems.
2. adding the theorems of the predefined errors.
3. proving that each error is unreachable, using an automatic theorem prover.
4. writing an error message for each unproven error.

Boogie stands out from other verification tools by three factors: Boogie supports two specification languages, which allows the expression of complex error definitions; is fit for system testing, finding errors on the program modules and their connections; and is fully automated, i.e. no user effort is needed, except for writing the specification and the developer postprocessing case discussed below.

However, full automation comes at a price. Since deciding whether an error is unreachable is an undecidable problem, Boogie is sure to emit false error messages at some programs. This results in posterior developer work, as the developer tries to see which messages refer to real errors and which messages are false.

Additionally, detecting errors in the entire program is not fit for unit testing, as most real-world programs take hours to fully analyze. Unit testing supports the module being developed at the moment. Thus, the feedback needs to be given as fast as possible.

3.2 Logic for Object Oriented Programming Tool

The Logic for Object Oriented Programming (LOOP) tool is an interactive static verification tool that finds errors in sequential Java programs semi-automatically. Being an interactive static verification tool, whenever LOOP fails to prove the absence of an error, LOOP requests new theorems from the user to complete the proof.

LOOP finds errors in a program by:

1. performing typechecking on the program and its specification.
2. translating the program and its specification into theorems.
3. proving that each error is unreachable, using an interactive theorem prover.

Like Boogie, LOOP defines custom errors using the specification language Java Modelling Language, and finds errors using a theorem prover. Unlike Boogie, LOOP uses an interactive theorem prover, instead of an automatic theorem prover.

An interactive theorem prover automates operations that are repetitive and error-prone, when done manually; like a calculator for program verification. LOOP's interactive theorem prover receives a set of theorems that represent the program and the program's specification and, with the help of the user, prove whether the program respects its specification.

However, interactive theorem provers cannot read Java classes. Instead, each interactive theorem prover has its own input language. LOOP further automates program verification by translating Java classes specified with JML to each interactive theorem prover's input language. LOOP supports both Isabelle and PVS interactive theorem provers.

3.3 Extended Static Checking for Java

The Extended Static Checker for Java (ESC/Java) [11] is a fast static verification tool and, as such, ESC/Java tries to find interesting errors as fast as possible. ESC/Java was developed at the Compaq Systems Research Center.

While most fast static verification tools check for a pre-defined set of common errors, ESC/Java can express new errors through the specification language JML.

For example, given a method specification that forbids the null argument and a program that calls that method with an explicit null argument, ESC/Java should be able to find the error.

Although JML is capable of expressing several properties (simple and complex), ESC/Java favors efficiency over finding all possible errors. Therefore, ESC/Java performs several simplifications to the program, losing information in the process.

ESC/Java finds errors in a program by:

1. simplifying the program, like ignoring overflow/underflow errors and unrolling loops up to a bounded size.
2. translating the simplified program into verification conditions.
3. proving that each error is unreachable, by supplying the verification conditions to the automatic theorem prover Simplify.
4. writing an error message for each unproven error. ESC/Java adds theorem information and source code information to the error message to help the developer fix the error.

ESC/Java takes less than five minutes, when checking for simple errors (passing a null argument, etc.) for programs between five hundred and a thousand lines of code. Studies on ESC/Java show that a programmer can annotate between three hundred and six hundred lines of code per hour on an existing unannotated program.

However, ESC/Java only supports errors on simple data as simplifying the program removes complexity, where other errors could be hidden. This poses an interesting trade-off on unit testing: remove complexity from the modules not under test, while keeping all the information of the module under test; or simplify everything to rely on fast feedback.

3.4 Korat: Automated Testing Based on Java Predicates

Korat [5] is an automated test case generation tool for Java programs that uses exhaustive testing to verify a program, i.e., tests a program with every possible input configuration up to a size.

Korat does not use symbolic nor concolic execution, but signals an important step in automated test case generation research.

Generated test cases that violate a method's precondition or the class' invariant are useless to testing, so Korat discards them. Korat supports both preconditions and class invariants through the specification language JML.

3.4.1 Example

To generate test cases, Korat needs both the interface of the input objects and a finitization method for each of the input objects. The finitization method bounds the input space of a given class to a size.

The following example describes a sample list object with a finitization method for the list head. The finitization method "finListHead(int)" tells Korat how to initialize each attribute and establishes a finitization object "f", a class domain - a set of objects of the same class - "items", and four field domains - a set of class domains for a given field - "head", "size", "value" and "next". For each field domain, the finitization object receives the corresponding class domain; the class domain "items" for "Item" and "Item.next", and the integer class domain for "size" and "Item.value".

The finitization method also receives an integer as a parameter. This integer defines the maximum number of objects in the class domain "items", and defines the upper limit of the field domains "size" and "Item.value".

All this information will be used, when creating objects of the class ListHead.

3.4.2 Architecture

To systematically explore the test case space, the authors of Korat instrumented Java's compiler, javac, to automatically generate a finitization method for each class. According to the authors of Korat, the generated finitization usually suffices (no user customization is needed). The instrumented javac also modifies each class's constructors, fields and preconditions to help the test case generation through backtrack pruning (discussed below).

Internally, Korat represents a test case as a candidate vector. Each position in the candidate vector represents the id of each field domain. Given the previous example, the candidate vector for the NULL test case (top-left in figure 3.1) is [0,0,0,0,0,0,0,0], where the input fields are ordered as "head", "size", 1.value, 1.next, 2.value, 2.next, 3.value and 3.next. Zero represents the value zero for the integer fields and the value NULL for the object fields.

Listing 3.1: Example of a Korat class finitization.

```
public class ListHead {
  private Item head;
  private int size;

  private class Item {
    int value;
    Item next;
  }

  public static Finitization finListHead(int NItems) {
    Finitization f = new Finitization(ListHead.class);
    ObjSet items = f.createObjects("Item", NItems);
    items.add(null);
    f.set("head", items);
    f.set("size", NItems);
    f.set("Item.value", NItems);
    f.set("Item.next", items);
    return f;
  }
}
```

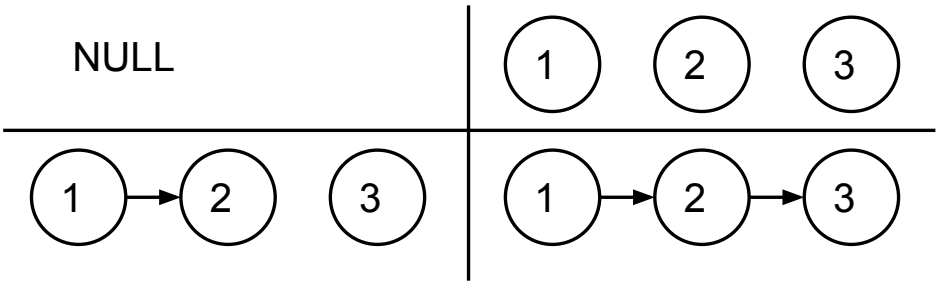

Figure 3.1: All possible graph combinations, given at most three objects of class Item (from the previous example) with one number each: NULL (top-left), [1] (top-right), [1,2] (bottom-left), and [1,2,3] (bottom-right). Korat ignores the number inside the item, so the combination [2,1,3] is equal to [1,2,3] and will not be created.

Listing 3.2: Example of a Korat class invariant.

```
public boolean repOk() {
  if (this.head == null)
    return this.size == 0;
  Item next = this.head.next;
  int size = 1;
  while (next != null) {
    size ++;
    next = next.next;
  }
  return this.size == size;
}
```

A naive test case space search would generate all possible candidate vectors, but Korat's search implements an optimization: backtrack pruning.

3.4.3 Backtrack Pruning

Backtrack pruning requires an executable precondition or class invariant. Consider the previous example has the following class invariant:

During the search, Korat does the following:

1. Initializes candidate vector at zeros.
2. Calls and monitors "repOk" to build an ordered list of field identifiers. Every time a field is accessed, an instrumented set method will add the field's identifier to the list, if the field was not added before.
3. Increments the field identifier of the last field in the candidate vector. If "repOk" returns true, then Korat stores the current candidate vector. Otherwise, Korat discards the test case.

The efficiency of backtrack pruning is only as good as "repOk". Precondition or class invariant implementations that return as soon as possible produce a more effective pruning than implementations that analyze

3.4.4 Shortcomings

Although Korat can generate several test cases in less than three seconds for simple cases, Korat is a limited tool. Korat only automatically initializes object graphs, so the user needs to specify how to initialize integers, booleans, strings, etc. Korat also has a fixed definition of isomorphism. On the other hand, JMLAutoTest [4] (a similar tool) has a customizable definition of isomorphism defined by the method equals.

Due to these limitations, Korat is not fit for the software development industry, but is instead a necessary step for more general tools.

3.5 ajmlc: The AspectJML Compiler

The AspectJML compiler enables specification contracts checking at run-time. To express the specification contracts, the AspectJML compiler uses AspectJML: an extension to Java Modeling Language (JML) focused on crosscutting specifications.

Listing 3.3: Example of an AspectJ advice. The custom java code is inserted after all methods that start with the word “set”.

```
after:execution(* Object+.set*(..)) {
  // custom java code.
}
```

Internally, the AspectJML compiler parses, typechecks and compiles java files annotated with AspectJML into the corresponding bytecode. Afterward, the AspectJML compiler inserts calls to a Run-time Assertion Checking (RAC) library in the generated bytecode using AspectJ.

AspectJ is a Java instrumentation framework that inserts code before, after or around certain methods. To do this, AspectJ defines pointcuts that depend on the desired methods' name, parameter types or return type. The pointcuts guide the instrumentation by defining which methods should change. The combination of a pointcut with the Java code to insert is called an advice (listing 3.3).

Additionally, RAC is a methodology to check whether the program respects its specification at run-time, for example, run-time assertion checking can instrument bytecode to insert assertion methods to every method call that throw an exception, whenever a method's precondition is violated (listing 3.4).

```
/*@
 @ requires x >= 0;
 @*/
public void setNumber(int x) {...}
```

```
before (Room room, int x):
  execution(* Room.setNumber(int))
  && this(room) && args(x) {
  RACChecker.checkPrecondition(room, x);
}
```

Listing 3.4: The code and specification on the left become the advice on the right.

3.5.1 Challenges

Both AspectJ and JML compilers face challenges when implementing methods that check specification violations at run-time, namely: Modular Reasoning, Crosscutting Specifications, and Specification as Documentation The AspectJML compiler sets itself apart from other specification compilers (whose source code is available at sourceforge.net) by partially solving these three problems.

Modular Reasoning Modular reasoning is the ability to reason about a class' objects using just the specification of that class and that class' supertypes. Using AspectJ to write a specification (listing 3.5) breaks modular reasoning, because the developer will need to review all AspectJ files before understanding which specification applies to the class under test.

However, JML writes the specifications in the modules themselves, unlike AspectJ, thus preserving modular reasoning.

Crosscutting Specifications Specification contracts (preconditions, postconditions, invariants...) are crosscutting concerns and work better when modularized with aspect-oriented programming.

Specifications in Design by Contract languages have limited ways to reuse preconditions or postconditions on several methods. Class invariants compact pre and postconditions expressions that do not rely on method-specific keywords (\result, \old, etc.) and are reused on all methods of the same visibility. In other cases, the pre or postcondition must be repeated and scattered among several methods.

However, AspectJ can merge several preconditions and postconditions into one advice, thus reducing the effort of writing crosscutting specifications.

Listing 3.5: Example of a postcondition contract using AspectJ. Note that there may be any number of postcondition contracts referencing the Room class scattered throughout the project.

```
after (Room room) returning (boolean hasChairs) :
  execution(boolean Room.hasChairs()) {
  if(!hasChairs) {
    throw new PostconditionViolatedException();
  }
}
```

Listing 3.6: AspectJML attaches pointcuts to crosscutting specifications, so that the developer writes the crosscutting specification only once.

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400;
@Pointcut("execution(* *Size(double, double)) && args(width,height)")
void setSize(double width, double height) {...}
void reSize(double width, double height) {...}
```

Specification as Documentation In JML, preconditions, postconditions and invariant declarations are placed directly in or next to the code being specified. When the specification is separated from the corresponding classes, the developer may not know about them and may break them inadvertently.

In AspectJ, the specification is separated from the code and does not contribute to system documentation. Additionally, an oblivious programmer can violate a method's pre and postconditions, when these are only recorded in aspects.

3.5.2 Solution

AspectJ modularizes crosscutting specifications, but does not support specification as documentation nor modular reasoning. JML supports specification as documentation and modular reasoning, but scatters crosscutting specifications across the file. AspectJML combines the benefits of AspectJ and JML by attaching pointcuts to JML specifications using @AspectJ.

In example 3.6, all methods ending with "Size" and with arguments "width" and "height" will have the crosscutting precondition.

This way, AspectJML maintains Modular Reasoning, Specification as Documentation, and solves specifications that crosscut one module. Specifications that crosscut several modules still need to be rewritten in each module to maintain Modular Reasoning.

3.5.3 Architecture

Like any compiler, the AspectJML compiler's architecture is structured in steps (figure 3.2):

1. parsing and typechecking Java files and corresponding AspectJML specification into an Abstract Syntactic Tree (AST);
2. generating a RAC AST from the old AST;
3. generating the AspectJ files from the RAC AST;
4. weaving the AspectJ files with the source code.

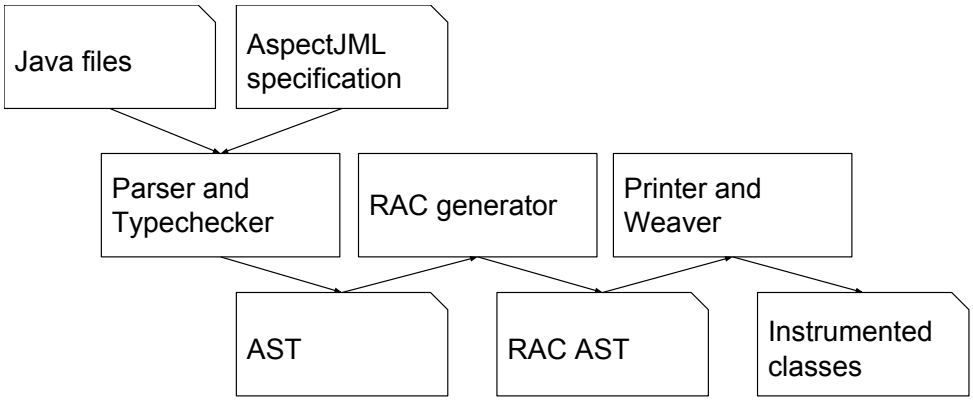

Figure 3.2: AspectJML compiler's steps to generate a RAC for a given project annotated with AspectJML. First, the AspectJML compiler parses and typechecks the source code and the specification into an AST. Then, the AspectJML compiler generates a RAC AST from the previous AST and, finally, writes to files and weaves the RAC AST with the source code.

The JML parser and typechecker of the AspectJML compiler are inherited from a different compiler, jmlc, which itself inherits from a custom Java compiler, Multijava.

3.5.4 Summary

The AspectJML enables run-time checking for specification contracts, but does not detect errors nor generate any test cases automatically by itself. Instead, the AspectJML uses other tools to generate test cases for the instrumented program, much like JUnit's test case execution framework.

3.6 CUTE: A Concolic Unit Testing Engine for C

CUTE [6, 17] is a concolic execution engine for C programs. There is also a version for Java called jCUTE [17], which handles concurrency problems like data races. Although CUTE's source code is no longer available, jCUTE's source code is available in github.

CUTE defines every unexpected program termination (segmentation fault, etc.) as an error. CUTE supports user-defined errors through assert functions, similar to JUnit [3]'s assert methods.

CUTE's syntactic evaluation implements the explicit path model checking algorithm. The explicit path model checking has the following steps:

1. Generate random input.
2. Execute the program with the previous input, until the program terminates or an error is found. If an error is found, store the input in a test case.
3. Collect all executed branches.
4. Negate one of the branches, for example, the branch var becomes $\neg var$.
5. Generate input from the new branches and repeat from the second step.

CUTE's constraint solving splits into integer constraints and pointer constraints. The integer constraint solving is implemented by lp_solve. Integer constraints are fed to lp_solve and lp_solve returns input values that respect the constraints. The pointer constraint solving is implemented by jCUTE using the pointer simplification optimization - described below - and equality constraint solving.

Before generating input, CUTE simplifies the collected branches with a few optimizations:

Listing 3.7: CUTE pointer simplification causes *index_pointer to be a different variable from *same_index_pointer. Therefore, *index_pointer will be evaluated as zero, instead of its real value, -1.

```
index_pointer = same_index_pointer;
*index_pointer = 0; *same_index_pointer = -1;
array[*index_pointer] = "index out of bounds";
```

Common sub-constraint elimination Common sub-constraint elimination removes all duplicate integer constraints before passing them to the arithmetic constraint solver.

Fast unsatisfiability check Fast unsatisfiability check looks for two syntactically negated constraints in the same constraint set, for example, having var and $\neg var$ in a conjunction.

The authors found that this optimization avoids 60% to 95% of the constraint solver calls in their case studies. This high percentage of symmetric constraints may derive from a high number of duplicate branches collected during the execution; After negating one of the collected branches - a step in the explicit path model checking algorithm -, the duplicate branch will become symmetric.

Incremental solving Incremental solving copies as much of the previous input as possible. In explicit path model checking, consecutively generated inputs are bound to be similar.

Pointer simplification removes all connections between pointer variables, for example, the pointer expression $a \rightarrow b \rightarrow c = 1$ is transformed into three expressions: $c = 1$, $*b = c$ and $*a = b$.

The explicit path model checking algorithm assures that the number of program executions is equal to the number of solvable program paths. However, other implementations of concolic execution [7, 10, 9] only execute the program, when a complex or unknown function is found, reducing the number of program executions.

Also, the pointer simplification optimization forces CUTE to miss some errors. Given the program below written in the C programming language, CUTE extracts the expressions $*index\_pointer = 0$ and $*same\_index\_pointer = -1$ ignoring that $*index\_pointer$ is actually -1.

3.7 Analysis on Related Work

Some of the mentioned tools tackle different problems - some tackle system testing, others unit testing - and complement each other. An ideal development environment would use ESC/Java, Korat and/or jCUTE several times throughout module development, and Boogie or LOOP once a day or once a week for system testing.

The data collected is not entirely useful for this comparison, because all tools were evaluated against different programs of varying size and complexity. Aspects like the time spent to find an error may be uncomparable, but aspects like user effort, specification used, and the types of errors that the tool can find are still useful, when comparing these tools.

The tools are evaluated against several aspects:

- Type of errors found.
- Time to find errors.
- User effort: expertise and automation.
- Output type: error messages versus test cases.

- Error definitions: Pre-defined versus user-defined.
- Specification type: Tool-specific versus reusable.

Table 3.1: Tool performance given the factors in the first row. All but LOOP are automated verification tools.

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

The LOOP tool gives the most feedback about errors in the program, but also requires the most user effort, requiring hours of interactive proving. Comparing with LOOP, all other tools require less user effort, because they are automatic verification tools. At most, the user needs to formally write the specification in the necessary specification language.

The best tool for automatically generating unit test cases will be CUTE/jCUTE, because CUTE handles the most error types in unit testing time, while Korat handles only pointer and boolean error types. Any other tool either takes too long for a unit testing environment, or detects only common errors.

Chapter 4

JMLCUTE

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

4.1 Goals

The verification tools described in chapter 3 either detect only simple errors (ESC/Java), lack specification support (CUTE), or cannot give feedback about errors within unit testing time (LOOP). JMLCUTE balances these three aspects optimizing them for unit testing in a Test-Driven Development.

4.1.1 Useful feedback about complex errors

It is important for JMLCUTE to focus on test case generation instead of error message generation.

When a verification tool gives feedback about an error, it is often in the form of error messages, giving the file name and line number of where the error was detected. However, some errors are more difficult to understand. For example, saying that a NullPointerException was thrown in a given method call (listing 4.1), while useful, does not tell the developer which data configuration triggered the exception. There are already tools that analyse the program for simple errors.

Since JMLCUTE aims to give feedback about complex errors, JMLCUTE favors test case generation over error message generation.

4.1.2 Automatically Generating Test Cases for Test-Driven Development

In Test-Driven Development, it is crucial to create test cases for the program even before the developer has wrote the first line of code. Test cases influence module development by simplifying modules: only

Listing 4.1: The complexMethod throws a NullPointerException at the shown program point. However, there is no information about the states of complexObject1 or complexObject2 which could help one fix the error.

```
ComplexClass result = complexMethod(complexObject1, complexObject2);
NullPointerException thrown at:
  complexMethod(ComplexClass,ComplexClass);
  ...
```

Listing 4.2: Example of an incomplete program with a complete specification.

```
//@ ensures \result == x + y;
public int add(int x, int y) {
  return 0;
}
```

write code to pass test cases; and providing feedback about program errors as soon as possible - the earlier an error is fixed, the smaller is the cost of fixing the error.

These two goals - simplifying module development and decreasing error fixing cost - put automated test case generation in an awkward position. Automated test case generation techniques have the potential to give feedback about program errors as early as possible, for example, executing a program with randomly generated input values may throw an uncaught exception. However, this kind of test cases gives no guidance to how the developer can simplify module development: one of the key aspects of Test-Driven Development.

The only way to simplify module development is by partially writing a specification against which the developer can test the program. If the developer chooses to specify the program using a formal specification language, the automated test case generation technique may be able to generate certain test cases based on the specification.

In the example below (listing 4.2), the method's implementation is incomplete, but the method's specification is complete. Some possibly generated test cases include $x = 0$ and $y = 0$ for correct behavior, and $x = 1$ and $y = 0$ for incorrect behavior.

Half of the errors found by verification tools that support formal specifications are specification errors. Therefore, adding specification support to Test-Driven Development will not only provide feedback on program errors, but also on design errors.

JMLCUTE is designed for Test-Driven Development and Design by Contract, such that, even when the developer is yet to write the first line of code, JMLCUTE can use the specification to generate test cases.

4.2 Architecture

JMLCUTE needs to detect complex errors in both the program and the specification and generate test cases for those errors in Test-Driven Development. From the verification techniques described in Chapter 3, the verification technique that is most capable of generating test cases for complex errors is concolic testing. Therefore, JMLCUTE is the combination of formal specification with concolic testing, deeply inspired in JML [19] and CUTE [6, 17].

4.2.1 First Architecture

JMLCUTE implements both a compiler for JML and a concolic testing engine.

The JML compiler's responsibility is to parse, typecheck and instrument project java files and their JML annotations/comments. The instrumentation inserts calls to the concolic testing library.

The concolic testing engine's responsibility is to analyze the program's bytecode and run-time execution and generate new test cases. To analyze the run-time of a program, the concolic testing engine instruments and simplifies the bytecode of the program to capture all methods' return values, if/for/while statements' condition expressions, among others.

A JMLCUTE execution consists of five steps (figure 4.1):

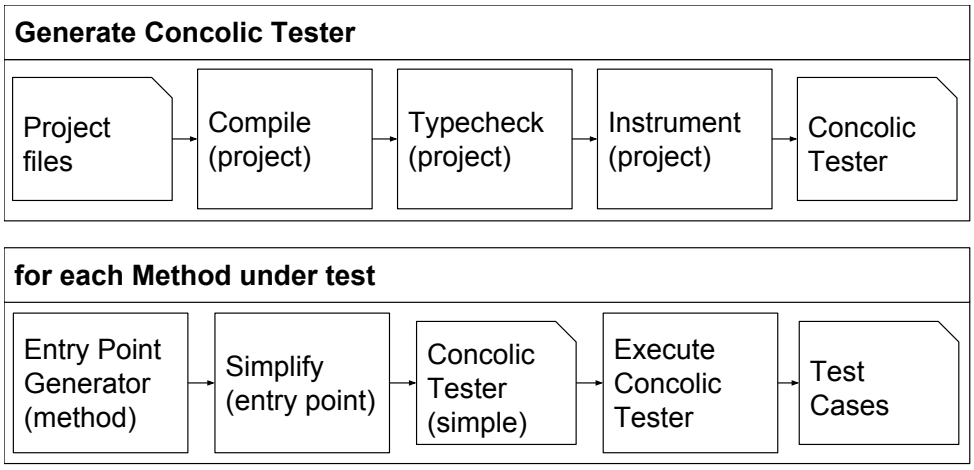

Figure 4.1: The first architecture of JMLCUTE. The first step is compiling the project and generating the concolic tester. The second step is generating an entry point (a main(String[]) method) per method under test, because the simplification of the concolic tester requires one entry point per method. Finally, the third step is concolically executing the concolic tester and generating test cases according to the concolic tester's output.

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 Instrument the compiled project files to detect errors of specification violation. This step is modified to insert a call to the concolic tester library per specification contract.

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.

An evaluation performed on this architecture demonstrated that most of the execution time is spent on repeatedly simplifying the concolic tester - approximately 9 minutes per method under test. In this case, a design modification may greatly improve JMLCUTE's performance.

4.2.2 Second Architecture

The first architecture scaled terribly when the number of methods under test increased, because most of the time was spent on simplifying the concolic tester. In Test-Driven Development, module implementation can only start after the test cases have been created, so decreasing the execution time of automated test case generation tools is especially useful.

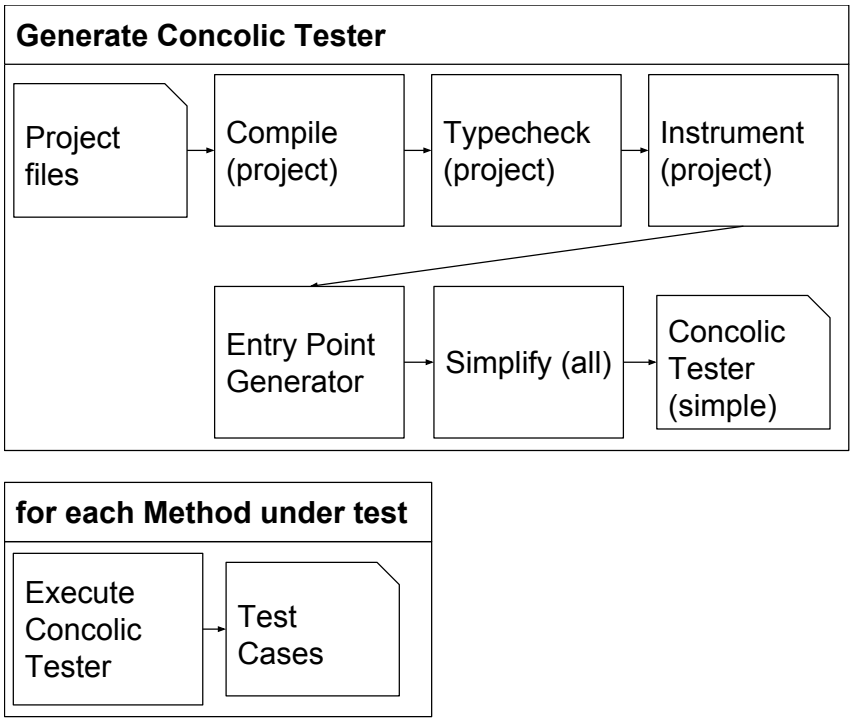

Figure 4.2: In the second architecture of JMLCUTE, the simplification of the concolic tester was compacted in one step. Instead of simplifying the concolic tester for every method, JMLCUTE firsts generates all needed entry points and simplifies them all resulting in drastically lower simplification time.

To decrease the amount of time spent simplifying the concolic tester, the simplification component was modified to accept several entry points at once, instead of just one (figure 4.2). This way, the second architecture compacts the simplification process in one step.

An evaluation performed on this architecture demonstrates that the execution time of the simplification step decreases from approximately 9 minutes per method under test to approximately 9 minutes per project compiled. This shows a decrease in complexity in the simplification step from $O(n * m)$ to $O(n)$, where n is the number of classes in the project and m is the number of methods under test. In the benchmark used, all projects use the same library which has greater program complexity than the project themselves. Consequently, most of the simplification time is spent on code common to all methods under test. Merging all simplification steps into one removes the time spent re-simplifying the common code, which gretly reduces the simplification time.

4.3 Using JMLCUTE

JMLCUTE's constraint solving engine imports a C library, lp_solve, through the Java Native Interface. Since its binaries are unavailable for all operating systems, this requires the user to compile the binaries on their own system following a set of rules.

Alternatively, the tool supports Vagrant to readily import a Linux 64-bit GUI-less virtual environment with an already compiled binary of the C library. Using vagrant, the user can type *vagrant up* at the root of the project to download and install the virtual machine (estimated download+install time: 1 hour) and *vagrant ssh* to connect to the machine.

The tool uses a bash script to automatically generate test cases for a given project; type *runjmlcute* <*fully qualified classname*>+ to automatically generate test cases for any number of classes. The script requires additional configuration, like source directories and concolic execution options. However, the

Listing 4.3: Example of test case generated by JMLCUTE. The comment shows that the test case does not violate the specification, but increases coverage.

```
/**
 * This test case increases coverage.
 */
public void test1() {
  i=0;
  input = new Object[3];
  simple.LinearArithmetic tmp1 = new simple.LinearArithmetic();
  input[i++] = tmp1;
  input[i++] = new Integer(170998211);
  input[i++] = new Integer(264470747);
  i=0;
  cute.Cute.input = this;
  jmlcute1.main(null);
}
```

Listing 4.4: Example for a precondition and postcondition contract for an addition method that only accepts strictly positive arguments.

```
class AddMath {
  //@ requires x > 0 && y > 0;
  //@ ensures \result == x + y;
  public int add(int x, int y) {
    return x + y;
  }
}
```

default configuration follows the Maven single-module standard directory layout, so no configuration is needed for projects using a single Maven module.

The script outputs the test cases (listing 4.3) and branch coverage information about the generated test cases. Each test case has a Javadoc comment explaining whether the test case causes a specification violation or simply increases branch coverage. Test cases without a comment about increasing coverage are created after a failure in the concolic engine's constraint solving to generate new input and are generated exclusively for debugging.

4.4 Java Modeling Language

JML [19] is a formal specification language that allows the specification of both correct and incorrect behavior for Java programs.

Developers often specify a program's behavior informally using Javadoc documentation, which is written in Java comments that start with the * symbol. Seeing the popularity of this specification technique, JML adapts Javadoc's syntax by writting the specification in Java comments that start with the @ symbol. An example of this syntax is present in listing 4.4

JML is focused on human readability and decreased required expertise, i.e. any Java programmer should be able to understand JML specifications. Therefore, the syntax within Java comments is also very similar to Java; JML is often expressed by a Java boolean expression with a few extra constructs that reference specification-based information.

Semantically, JML implements many aspects of the Design by Contract development process described in chapter 2: preconditions, postconditions, invariants, among others. Most contracts are ex-

pressed by a keyword followed by a Java boolean expression, for example, preconditions are identified by the keyword *requires*, postconditions by the keyword *ensures*, and invariants by the keyword *invariant* as can be seen in listing 4.4.

4.4.1 Purity Analysis

JML also features purity analysis. Since much of JML's syntax can be used directly in assert functions and assert functions should not have side-effects, JML offers a notion of side-effect free methods: the notion of purity. A method is pure, when the method exclusively calls pure methods and does not assign any value to an object that existed prior to the method call. Pure methods can assign new values to their arguments, but not to the attributes of their arguments.

Unfortunately, the notion of purity has limited usefulness when writing a specification, for example, the notion of purity does not allow the comparison of complex objects in the specification, because creating complex objects usually requires methods that alter the state of the object and are, therefore, impure.

Chapter 5

JMLCUTE Implementation

Chapter 4 described JMLCUTE's architecture and mentioned a few of its components. This chapter explains each of JMLCUTE's components extensively, in particular, the JML parser, typechecker and instrumenter, the simplification component, the concolic testing engine, and the branch coverage tool.

For each component, this chapter discusses goals, challenges, and architecture, among other component-specific topics.

5.1 Parser, Typechecker and Instrumenter for the Java Modelling Language

The parser, typechecker and instrumenter for JML used in JMLCUTE is built upon the source code of another JML compiler, the AspectJML compiler [12] (described in chapter 3).The AspectJML compiler is distributed with the Apache v2 License.

5.1.1 Overview

JMLCUTE's main goal for the JML parser, typechecker and instrumenter is to enable JML understanding on a concolic testing engine. The modified AspectJML compiler's RAC methodology and JML parsing allows a concolic testing engine to evaluate each specification contract at run-time and detect an error whenever the specification contract is violated. AspectJML compiler also enables the use of @AspectJ annotations that may help the developer write a more modular, readable specification.

AspectJML is an extension to JML focused on crosscutting specifications used in the AspectJML compiler. Therefore, the AspectJML compiler parses, typechecks and compiles java files annotated with AspectJML into the corresponding bytecode. The AspectJML compiler also inserts calls to a RAC library in the program's bytecode using AspectJ.

5.1.2 Changes to AspectJML

Unfortunately, the RAC library used by the AspectJML compiler does not automatically generate test cases; the library only prints error messages, if an execution triggers the error.

To generate test cases, we replaced the RAC library calls with calls to a concolic testing engine (listing 5.1, explained in detail later in this chapter). To do this, JMLCUTE modifies each aspect generated by the AspectJML compiler, such that every RAC library call is replaced by the correct concolic

Listing 5.1: Generated advice to check the normal postcondition of method Room.hasChairs.

```
after (Room room) returning (boolean hasChairs) :
  execution(boolean Room.hasChairs()) {
    Cute.Assert(!hasChairs);
}
```

Listing 5.2: Example of an entry point that calls the method `guard` with the input objects `arg0` and `arg1`. The example has been simplified by removing casts and import statements.

```
public static void main(String[] args) throws Throwable {
  Conjunction receiver = Cute.input.Object("Conjunction");
  Cute.Assume(receiver != null);
  ArrayList arg0 = Cute.input.Object("ArrayList");
  Integer arg1 = Cute.input.Integer();
  receiver.guard(arg0, arg1);
}
```

testing library call - Cute.Assert or Cute.Assume. Now, instead of simply printing an error message, the generated advice calls a concolic testing engine to generate new test cases.

5.2 Entry Point Generator

JMLCUTE's entry point generator is a custom built component. The entry point generator's responsibility is to generate one entry point class per method under test.

In concolic testing, the program is instrumented and executed, such that each execution receives a different input.

Several problems arise from this statement. First, a Java program can only be executed through the `public static void main(String[])` method - the Java entry point -, and the method under test is rarely the main method. Secondly, the concolic testing engine needs to recognize which variables are input and which variables are not. Finally, the concolic testing engine needs to instantiate the input variables.

JMLCUTE solves the Java entry point problem by printing and compiling an entry point class (listing 5.2) that contains a main method that, in its turn, instantiates the receiver object and invokes the method under test on that receiver object. If the method under test has more than zero arguments, the entry point class will also need to instantiate the arguments. For now, JMLCUTE only concolically tests public non-static methods, but JMLCUTE can be easily extended to accept other types of methods.

JMLCUTE solves the second problem mentioned above - recognizing which variables are input and which variables are not - by assuming that only the arguments of the method under test are input variables, for example, the input variables of an `add(int,int)` method would be the two integer arguments. Then, JMLCUTE marks the method arguments as input variables by instantiating them through an input library offered by the `concolic.input` package of jCUTE.

Automatically instantiating receiver objects and arguments looks simple at first, but hides great complexity, for example, the class to instantiate may have no public constructors (listing 5.3), or the only public constructors available may have complex arguments of their own. Concolic testing adds another layer of complexity: the instantiated objects must respect the generated test case. If the instantiated objects differ from the intended test case, the concolic execution fails and may miss errors that could otherwise be found.

Listing 5.3: Example of a class that cannot be instantiated.

```
public class Uninstantiable {
  private Uninstantiable() {}
}
```

Unfortunately, the input library offered by jCUTE uses a limited number of constructors. If the class to instantiate is a primitive type (int, boolean, among others) or a primitive wrapper type (Integer, Boolean, among others), the input library instantiates the object by calling the correct one-argument constructor. For all other classes, the input library can only instantiate the object, if the class provides a constructor with zero arguments. If the class has no constructors with zero arguments, the input library fails to instantiate the object, and the test case generation fails.

5.2.1 Summary

The entry point generator partially solves each problem by:

Java entry point Generating an entry point class with a public static void main method that calls the method under test.

Detecting input variables Using special instantiation methods provided by the input library.

Object instantiation Calling the correct constructor for primitives or primitive wrapper types, or calling the constructor with zero arguments for all other classes.

5.3 Simplification Component

JMLCUTE's simplification component is provided by jCUTE's `instrument` package. The simplification component's responsibility is to simplify the program's bytecode and help the capture of symbolic expressions by the concolic testing engine.

To capture arithmetic, boolean and pointer constraints, JMLCUTE needs to recognize all additions, subtractions and assignments (among others) present in the program's bytecode. Unfortunately, Java bytecode is a complex language with over 190 possible operations (opcodes) [1]. Therefore, extracting symbolic constraints from Java bytecode is a daunting task.

JMLCUTE overcomes this complexity by simplifying Java bytecode into a simple high-level language, Jimple, using the SOOT framework [2]. Once the bytecode is simplified into a Jimple AST, the simplification component inserts concolic testing library calls before or after an even smaller subset of Jimple's statements and expressions:

Instrumented Statements assignment, `if`, invoke, monitor, `return` and switch.

Instrumented Expressions Arithmetic operations, array initializations, casts, constants, invocations (static, non-static, and array and string length methods), local variables and references (array, static or non-static).

The concolic testing library's resposibility is to store symbolic expressions. To do this, the concolic testing library creates and stores symbolic expressions of each object in an hash map. For example, for the assignment `Integer x = 2 + 2`, the concolic testing library:

[1] http://docs.oracle.com/javase/specs/jvms/se8/html/index.html
[2] http://sable.github.io/soot

1. Creates an arithmetic expression that symbolizes the addition of 2 with 2,

2. Gets the identity hashcode of the Integer object `x`,

3. Stores the arithmetic expression in an hash map with the hashcode as the key.

Besides instrumenting the aforementioned expressions and statements, the simplification component captures additional information to guide components other than the concolic testing library. The simplification component calculates the total number of branches of each method and instruments all `if`, `for`, among other branching statements to count the number of branches covered at run-time in a given concolic execution. The branch coverage tool (present in this chapter) later uses this information to evaluate the output of the concolic testing.

The simplification component also helps the error detection library by signaling all uncaught exceptions at the top-level - the main method - as program errors. To do this, the simplification component wraps the main method in a `try-catch` and, depending on the type of exception caught, either ends the concolic execution with an error message, or an assumption violated message.

5.3.1 Changes to the Simplification Component

As stated earlier, the simplification component needs to wrap the Java entry point - the main method - in a `try-catch`. Unfortunately, the original simplification component only accepted one main method per simplification. This problem leads to a greater than linear increase of time, when concolic testing more than one method, as seen in Chapter 6.

To mend this situation, a second architecture of JMLCUTE modifies the simplification component by accepting several entry points, instead of one. More precisely, we modified the error detection of the simplification component by wrapping all given entry points with a `try-catch`. Since all entry points generated by the entry point generator are independent of one another, there is no risk of ending the concolic execution prematurely.

5.3.2 Summary

The simplification component transforms the Java bytecode in a easy to instrument Jimple language. Then, the simplification component instruments every symbolic expression and branching statement, counts the total number of branches of each method, and detects all uncaught exceptions at the entry point level.

The original simplification component can only detect uncaught exceptions on one entry point. We modify the simplification component to detect uncaught exceptions on any number of entry points.

5.4 Concolic Testing Engine

JMLCUTE's concolic testing engine is provided by jCUTE. The concolic testing engine's responsibility is to provide library calls to the concolic tester that store symbolic expressions, detect errors, and predict new program paths by solving the symbolic expressions.

The concolic testing engine consists of three smaller components: the concolic testing library, the error detection library, and the path finder (figure 5.1).

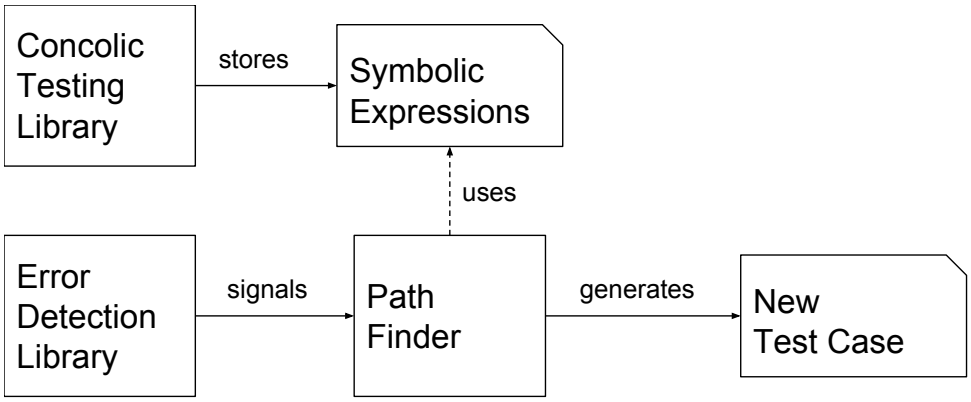

Figure 5.1: Architecture of the concolic testing engine. The concolic testing engine consists of three smaller components: the concolic testing library that stores symbolic expressions about the run-time behavior of the method under test, the error detection library that signals the path finder when a given condition is false, and the path finder that uses the stored symbolic expressions to generate new test cases.

5.4.1 Concolic Testing Library

The concolic testing library is provided by jCUTE (`Call` class in the `cute.concolic` package) and its responsibility is to capture and store symbolic expressions.

The simplification component (described in this chapter) inserts concolic testing library calls in the concolic tester, such that, whenever an arithmetic, boolean or pointer expression (among others) is evaluated, the concolic testing library receives enough information about the run-time execution of the program to generate and store the corresponding symbolic expressions.

Internally, the concolic testing library works like a stack (figure 5.2). Every time the concolic tester evaluates an expression or operation, the concolic testing library receives the arguments and the type of the operation from the concolic tester and pushes them into a stack.

Later, when the concolic tester evaluates an assignment or a branching statement, the concolic testing library pops the previously stored argument and operation type information to create a new symbolic expression. Then, the concolic testing uses the address of the object being assigned to as a key to store the new symbolic expression in a hash map.

5.4.2 Error Detection Library

Like any verification tool, the concolic testing engine needs to detect the presence of errors. JMLCUTE expresses errors through two assertion methods inherited from jCUTE: `Cute.Assert(boolean)` and `Cute.Assume(boolean)`.

Cute.Assert `Cute.Assert`'s argument is a boolean expression that represents a property that must be respected at run-time. If the boolean argument is evaluated to false at run-time, the property has been violated and `Cute.Assert` signals the path finder that an error has occurred.

Cute.Assume Similar to `Cute.Assert`, `Cute.Assume`'s argument is a boolean expression that represents a property that must be respected at run-time. However, unlike `Cute.Assert`, violating `Cute.Assume`'s property does not cause an error. Instead, the property violation represents a flaw in the assumptions of concolic testing, such that any test case generated based on this flaw is useless.

For example, when concolically testing the square root method `double squareRoot(double x)` (listing 5.4), we can assume that $x \geq 0$, because the square root of a negative number is a

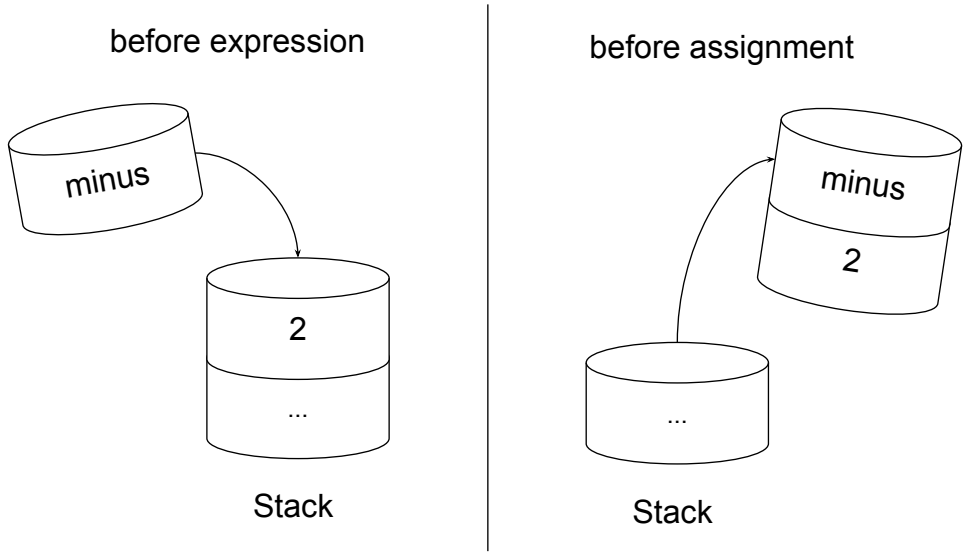

Figure 5.2: Example of the use of the stack to create a symbolic expression In the left side, before executing the minus expression, the concolic testing library pushes the operation and its argument to the stack. In the right side, before an assignment uses the expression, the concolic testing library pops the expression and its arguments to create a symbolic expression.

Listing 5.4: Example of a possible assumption. If `squareRoot` is the method under test, generating test cases in which x is lesser than zero is useless.

```
//@ requires x >= 0;
public double squareRoot(double x) {
...
}
```

complex number that cannot be represented by the type `double`. If JMLCUTE generates a test case in which the `squareRoot` behaves incorrectly, but $x < 0$, the test case is not helping the developer fix any error, because the developer is not expecting `squareRoot` to behave correctly, when $x < 0$, in the first place.

Using both assertion methods, JMLCUTE can decide which test cases are interesting and which test cases are useless.

The original error detection library only has the exit code functionality of `Cute.Assert`. JMLCUTE modifies the error detection library to add exit code functionality to `Cute.Assume`, i.e., the exit code has a bit reserved for `Cute.Assume`. If the concolic tester ended in an assumption violation, the bit is one. Otherwise, the bit is zero.

5.4.3 Path Finder

The path finder is provided by jCUTE and its responsibility is to:

1. modify the stored symbolic expressions;

2. solve the resulting constraints.

When the path finder is signaled by the error detection library that an error has occurred, the path finder must first split the symbolic expressions in arithmetic expressions and pointer expressions. Then,

the path finder's constraint solves each group using two different constraint solvers: the arithmetic solver and the pointer solver.

While the pointer solver is part of jCUTE, the arithmetic solver is part of lp_solve [20], a linear arithmetic constraint solver developed in the C programming language. The path finder uses lp_solve by using the Java Native Interface, which allows the use of C libraries in Java.

5.4.4 Changes to the Concolic Testing Engine

The concolic tester needs to execute like a real Java application, so JMLCUTE encapsulates the concolic tester in a process. Like any Java process, the concolic tester terminates with an exit code that can be captured by the process that started the concolic tester. Exit codes are usually automatically generated by the Java Virtual Machine (JVM), but can be customized to contain information about the exit state of a Java process.

The concolic tester inherited some exit code functionality from jCUTE; when the concolic tester is about to terminate, the concolic tester detects what triggered the termination - be it a deadlock, a data race, a tool error, or a full branch coverage - and adds it to the exit code in bit-vector style. JMLCUTE adds the "specification violated" code, the "assumption violated" code and the "coverage increased" code to the exit code functionality.

Additionally, the jCUTE's concolic testing engine does not generate test cases files. Instead, the jCUTE's concolic testing engine stores information about the generated test cases and offers methods that create a JUnit [14] file with the generated test cases.

Later, JMLCUTE uses these methods and the concolic tester's exit code to decide whether or not to create the new JUnit file or to provide each new test case with an explanatory comment: if the generated test case violates the specification, JMLCUTE writes a comment before the test case that reads "This test case causes a specification violation.", if the generated test case increases branch coverage, JMLCUTE writes a comment that reads "This test case increases coverage.".

5.5 Branch Coverage Tool

The branch coverage tool used in JMLCUTE is already present in jCUTE (in the `BranchCoverageLog` class in the `cute.logging` package). The branch coverage tool's responsibility is to evaluate how useful the test cases generated by JMLCUTE are.

JMLCUTE evaluates the generated test cases by capturing:

- the number of executed branches for each method invoked.
- the percentage of executed branches over the total number of branches.
- the number of functions invoked.
- the number of threads used during concolic execution; useful when detecting data races.
- the numbers of iterations per concolic testing.
- the date at which the program was concolically tested (utility purposes).

The branch coverage tool was modified to write the coverage information of each method under test to a file (listing 5.5). The file contains all the information described above.

Listing 5.5: Branch coverage information about the test cases generated for a given method.

```
-------------------------------------------------------------------
Printing branch coverage statistics
-------------------------------------------------------------------
2 branches covered out of 2 branches in the function <SocialEventPlanner.events.jmlcute1: void
    main(java.lang.String[])>
4 branches covered out of 4 branches in the function <Util.Utilities: void <init>(int)>
2 branches covered out of 2 branches in the function <eventb_prelude.BSet: void
    <init>(java.util.TreeSet)>
2 branches covered out of 2 branches in the function <eventb_prelude.Enumerated: void
    <init>(int,int)>
Total functions invoked = 4
Total branches covered = 10
Percentage of branches covered = 100.0 in time 1442330201552 (ms)
Number of threads = 1
Number of iterations = 5
-------------------------------------------------------------------
```

5.6 Implementation Summary

JMLCUTE's implementation is focused on integrating the AspectJML compiler with the concolic testing engine jCUTE. To do this, JMLCUTE:

1. Modifies all RAC calls in the AspectJML compiler to concolic testing library calls,
2. Generates an entry point class for each method under test,
3. Modifies the simplification component to enable multiple entry point simplification,
4. Modifies the concolic testing engine to improve the exit code functionality.
5. Wraps the concolic tester in a process,
6. Uses the concolic tester's exit code to decide whether to create a JUnit file to the generated test case,
7. Logs all branch coverage information about each concolically executed method in a file.

Chapter 6

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 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.

6.1 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.

6.2 Setup

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

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.

Listing 6.1: The `add(int,int)` method of the `LinearArithmetic` class.

```
/*@
 @ requires x > 0 && y > 0;
 @ assignable \nothing;
 @ ensures \result == \old(x) + \old(y);
 @ signals(Exception) false;
 @ also
 @ requires x <= 0 || y <= 0;
 @ assignable \nothing;
 @ ensures false;
 @ signals(Exception) true;
 */
public int add(int x, int y) throws Exception {
  return x + y;
}
```

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

6.3 Samples Benchmark

The Samples benchmark is custom made and consists of several classes, each one demonstrating one type of constraints. Every class in the Samples benchmark consists of only public non-static methods.

We used the second architecture of JMLCUTE to evaluate the Samples benchmark.

6.3.1 Description

The classes present in the Samples benchmark are:

ConjunctionOnCollection The `ConjunctionOnCollection` class provides several methods that contain constraints on collections, i.e. classes similar to the classes of the Java Collection framework. The collections used include `BSet`, a complex implementation of a `TreeSet`, `CustomBSet`

ConjunctionOnIndirectCollection The `ConjunctionOnIndirectCollection` class mimics the `ConjunctionOnCollection` class by providing methods with constraints on collections. However, all collections beside the arguments of the methods are hidden behind a private attribute.

LinearArithmetic The `LinearArithmetic` class provides two methods that contain linear arithmetic constraints.

NotNull The `NotNull` class provides methods that contain non-null pointer constraints. These methods test whether JMLCUTE can generate objects using the constructor with zero arguments.

The Samples benchmark contains several specification errors, for example, the specification error in the `add(int,int)` method of the `LinearArithmetic` class (listing 6.1). The `add` method should throw an exception, when at least one of the arguments is negative, but the implementation violates this property by always returning the addition of both arguments.

6.3.2 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 6.1) can be misleading. While JMLCUTE has a lower branch coverage 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, for example, the `aspectOf()` method that is present in every aspect file.

Table 6.1: The branch information gathered from the evaluation of the Samples benchmark. The first row states a code for each of the methods verified:
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)`.

	A	B	C	D	E	F	G	H	I
Total branches, jmlcute	20	34	14	4	14	18	10	12	14
Total branches, jcute	8	22	6	2	16	2	4	4	4
Branches covered, jmlcute	10	13	7	3	6	4	8	10	11
Branches covered, jcute	6	9	3	2	5	2	4	4	4
Branches covered (%), jmlcute	50	38,2	50	75	42,9	22,2	80	83,3	78,6
Branches covered (%), jcute	75	40,9	50	100	31,3	100	100	100	100

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.

6.4 Event-B Benchmark

The Event-B benchmark is an edited version of a benchmark available at the EventB2Java Rodin plug-in webiste [1]. Like the Samples benchmark, only the public non-static methods were evaluated.

Each of this benchmark's paths is highly dependent on collection constraints (related to the Java Collection framework). Therefore, this benchmark tests JMLCUTE's and jCUTE's ability to capture and solve complex pointer constraints.

6.4.1 Description

The benchmark used is edited from the output of an Event-B to JML translation project [21].

The benchmark consists of 109 classes divided into nine projects:

binary_search_sequential Implements binary search.

linear_search_sequential Implements linear search.

min_array Finds the minimum element in an array of integers.

[1] `http://poporo.uma.pt/~ncatano/Projects/EventB2Java/EventB2Java_studies.html`

MIO_multi_threaded Mass transportation system that includes buses following the main routes or outskirts of a city.

reversing_array Reverses an array.

SM03_multi_threaded A simple state machine.

SocialEventPlanner_multi_threaded Renamed SocialEventPlanner. Implements some of the functionality expected of a social network: adding comments, replying to comments, authorization constraints, among others.

sorting_array Sorts an array.

sqrt_number Calculates the square root of a number.

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, manually added auto-boxing, and removed some of the specification invariants and history constraints.

6.4.2 First Evaluation Results

In the first evaluation of the Event-B benchmark, we use the first JMLCUTE architecture, in which the project is instrumented every time an entry point is generated. In this iteration, we also exclusively evaluate 30 methods of the `SocialEventPlanner` project.

Both JMLCUTE and jCUTE took six hours to complete the concolic testing of the 30 methods. It is important to note that most of the time was spent on simplifying the generated entry points. The excessive amount of execution time lead to the modification of JMLCUTE's architecture and a second evaluation phase.

Once again, JMLCUTE sees an expected decrease in the percentage of branches covered(table 6.2), when comparing JMLCUTE with jCUTE. However, the number of branches covered doubled, hinting that a powerful enough constraint solver could solve all added constraints.

Table 6.2: Branch coverage information about the first evaluation of the Event-B benchmark.

	Total branches	Branches covered	Methods invoked
JMLCUTE	8440	2872 (34%)	1348
jCUTE	2664	1375 (52%)	619

All 30 evaluated methods contain mostly collection constraints. Since neither JMLCUTE nor jCUTE are capable of solving collection constraints, neither JMLCUTE nor jCUTE were capable of generating more than one or two test cases per method.

6.4.3 Second Evaluation Results

In the second evaluation of the Event-B benchmark, we use the second JMLCUTE architecture, in which the project is instrumented after all entry points are generated.

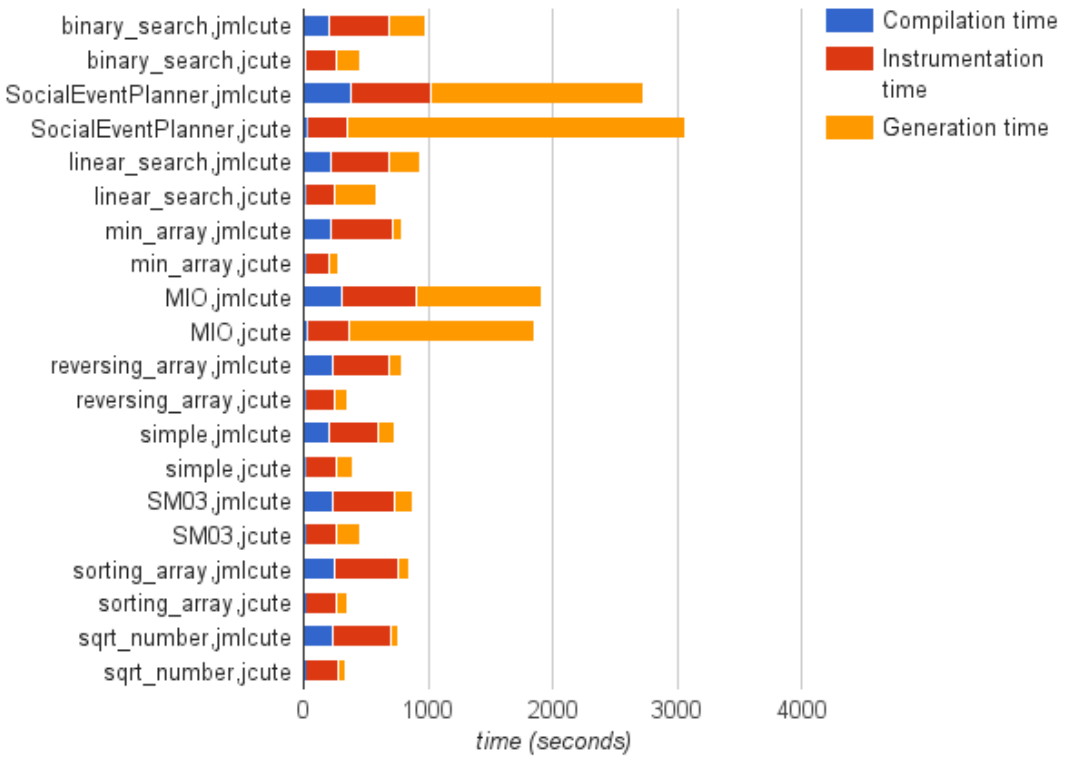

Figure 6.1: The execution times of both jCUTE and JMLCUTE for each project of the Event-B benchmark. The times measured were compilation time - time spent in the AspectJML compiler for JMLCUTE, or javac for jCUTE -, instrumentation time - time spent in the simplification component -, and generation time - which includes concolic testing and test case printing.

The results show a significant reduction in execution time, when comparing with the results of the first evaluation (figure 6.1). While generating test cases for 30 public methods took six hours in the first architecture, generating test cases for all 102 public methods of the *SocialEventPlanner* package took only 45 minutes in JMLCUTE's second architecture.

The evaluation on the smaller projects show 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.

When comparing the execution times of both JMLCUTE and jCUTE to the total number of branches of each project (figure 6.2), 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 the creation of the initial state of a test case violates the specification. For example, in the MIO_multi_threaded project, some constructor calls violate the class's specification. When JMLCUTE detects that these constructor calls violate the class's specification, JMLCUTE terminates execution. On the other hand, jCUTE does not evaluate the class's specification and, thus, does not terminate before executing the method under test. Therefore, JMLCUTE terminates faster than jCUTE in certain test cases reducing the overall generation time of the MIO_multi_threaded project - among others.

In terms of the number of errors found, neither tool found any error, because both tools perform poorly on programs with complex pointer constraints, i.e. collection constraints.

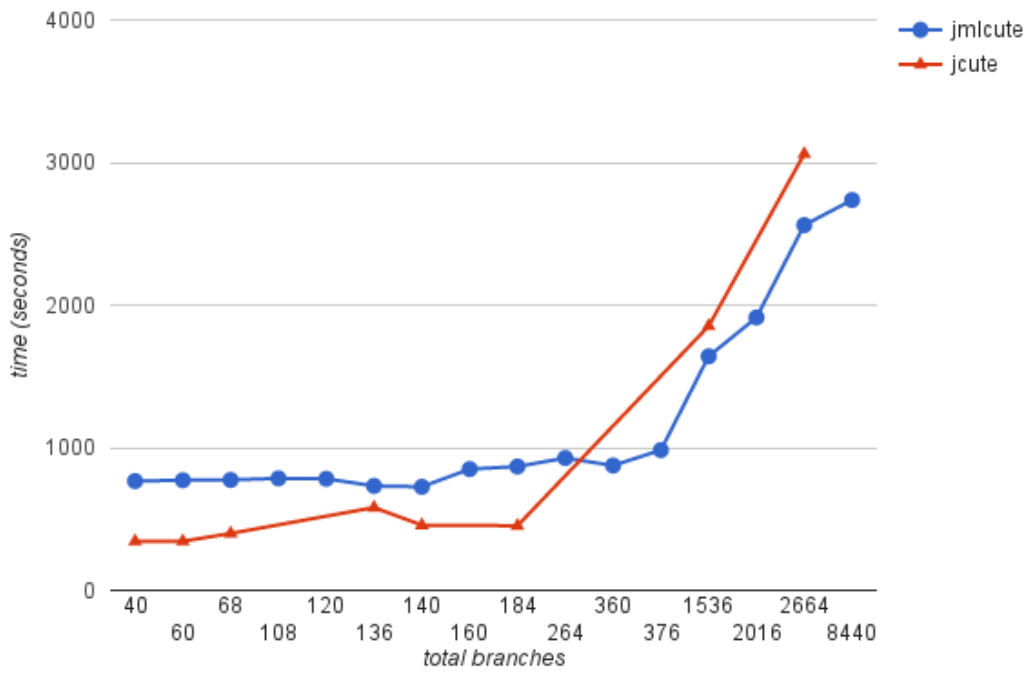

Figure 6.2: Execution times related to the total number of branches of each project for both JMLCUTE and jCUTE. The number of branch total instances is greater than the number of projects, because JMLCUTE and jCUTE generate a different number of branch totals for each project. Note that the horizontal axis is not scaled linearly.

6.5 Evaluation Summary

JMLCUTE was capable of concolically testing 109 classes in approximately the same execution time as jCUTE, outperforming jCUTE on complex projects.

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.

Chapter 7

Conclusion

This dissertation 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.

7.1 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 [22] 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.

Bibliography

[1] P. Godefroid, P. De Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin, "Automating software testing using program analysis," *Software, IEEE*, vol. 25, no. 5, pp. 30–37, 2008.

[2] L. Clarke *et al.*, "A system to generate test data and symbolically execute programs," *Software Engineering, IEEE Transactions on*, no. 3, pp. 215–222, 1976.

[3] K. Beck and E. Gamma, "Test infected: Programmers love writing tests," *Java Report*, vol. 3, no. 7, pp. 37–50, 1998.

[4] G. Xu and Z. Yang, "Jmlautotest: A novel automated testing framework based on jml and junit," in *Formal Approaches to Software Testing*, pp. 70–85, Springer, 2004.

[5] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," in *ACM SIGSOFT Software Engineering Notes*, vol. 27, pp. 123–133, ACM, 2002.

[6] K. Sen, D. Marinov, and G. Agha, *CUTE: a concolic unit testing engine for C*, vol. 30. ACM, 2005.

[7] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, pp. 213–223, ACM, 2005.

[8] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal methods for Components and Objects*, pp. 364–387, Springer, 2006.

[9] X. Deng, J. Hatcliff, *et al.*, "Kiasan/kunit: Automatic test case generation and analysis feedback for open object-oriented systems," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pp. 3–12, IEEE, 2007.

[10] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, pp. 209–224, 2008.

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Pldi 2002: Extended static checking for java," *ACM Sigplan Notices*, vol. 48, no. 4S, pp. 22–33, 2013.

[12] H. Rebêlo, G. T. Leavens, M. Bagherzadeh, H. Rajan, R. Lima, D. M. Zimmerman, M. Cornélio, and T. Thüm, "Aspectjml: Modular specification and runtime checking for crosscutting contracts," in *Proceedings of the 13th international conference on Modularity*, pp. 157–168, ACM, 2014.

[13] K. Sen, "Concolic testing," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 571–572, ACM, 2007.

[14] E. Gamma and K. Beck, "Junit: A cook's tour," *Java Report*, vol. 4, no. 5, pp. 27–38, 1999.

[15] C. Beust and H. Suleiman, *Next Generation Java Testing: TestNG and Advanced Concepts*. Pearson Education, 2007.

[16] B. Meyer, “Applying’design by contract’,” *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[17] K. Sen and G. Agha, “Cute and jcute: Concolic unit testing and explicit path model-checking tools,” in *Computer Aided Verification*, pp. 419–423, Springer, 2006.

[18] M. Huisman, *Reasoning about Java programs in higher order logic using PVS and Isabelle*. [SI: sn], 2001.

[19] G. T. Leavens, A. L. Baker, and C. Ruby, “Preliminary design of jml: A behavioral interface specification language for java,” *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1–38, 2006.

[20] M. Berkelaar, K. Eikland, P. Notebaert, *et al.*, “lpsolve: Open source (mixed-integer) linear programming system,” *Eindhoven U. of Technology*, 2004.

[21] V. Rivera and N. Cataño, “Translating event-b to jml-specified java programs,” in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pp. 1264–1271, ACM, 2014.

[22] C. Prud’homme, J.-G. Fages, and X. Lorca, *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014.

Appendix A

Appendix Evaluation Tables

This appendix presents the results for the evaluation of each project and for each tool (jCUTE and JMLCUTE). This appendix shows the branch total, branches covered, functions invoked, percentage of branches covered, and concolic iterations used in each public non-static method.

Table A.1: Evaluation table for the project binary_search_sequential with the second architecture.

Tool	Method under test	Total branches	Covered branches	Functions invoked	Covered branches (%)	Iterations
jmlcute	dec.guard_dec	45,99972741	27	15	58,696	3
	dec.run_dec	45,99972741	27	15	58,696	3
	found.guard_found	70,0005122	41	20	58,571	3
	found.run_found	70,0005122	41	20	58,571	3
	inc.guard_inc	72,04116638	42	21	58,3	3
	inc.run_inc	72,04116638	42	21	58,3	3
Totals		376,082812	220	112		
Compilation time	205					
Instrumentation time	478					
Generation time	300					
User time	891					
Sys time	64					
jcute	dec.guard_dec	18,00554017	13	4	72,2	4
	dec.run_dec	18,00554017	13	4	72,2	4
	found.guard_found	25,99984706	17	6	65,385	4
	found.run_found	27,99987556	18	7	64,286	4
	inc.guard_inc	24,00112946	17	6	70,83	4
	inc.run_inc	26,00028889	18	7	69,23	4
Totals		140,0122213	96	34		
Compilation time	8					
Instrumentation time	254					
Generation time	195					
User time	415					
Sys time	33					

Table A.2: Evaluation table for the project linear_search_sequential with the second architecture.

Tool	Method under test	Total branches	Covered branches	Functions invoked	Covered branches (%)	Iteration
jmlcute	found.guard_found()	66,03773585	35	18	53	3
	found.run_found()	66,03773585	35	18	53	3
	progress.guard_progress()	66,03773585	35	18	53	3
	progress.run_progress()	66,03773585	35	18	53	3
Totals		264,1509434	140	72		
Compilation time	214,5					
Instrumentation time	477,75					
Generation time	235,25					
User time	848,5					
Sys time	56					
jcute	found.guard_found()	34,00309119	22	8	64,7	4
	found.run_found()	35,9375	23	9	64	4
	progress.guard_progress()	32	22	8	68,75	4
	progress.run_progress()	33,9985218	23	9	67,65	4
Totals		135,939113	90	34		
Compilation time	15					
Instrumentation time	229					
Generation time	336,5					
User time	608					
Sys time	31					

Table A.3: Evaluation table for the project min_array with the second architecture.

Tool	Method under test	Total branches	Covered branches	Functions invoked	Covered branches (%)	Iterations
jmlcute	dec.guard_dec()	18	9	6	50	3
	dec.run_dec()	18	9	6	50	3
	inc.guard_inc()	18	9	6	50	3
	inc.run_inc()	18	9	6	50	3
	mini.guard_mini()	18	9	6	50	3
	mini.run_mini()	18	9	6	50	3
Totals		108	54	36		
Compilation time	223,5					
Instrumentation time	493,5					
Generation time	68					
User time	705					
Sys time	57					
jcute	dec.guard_dec()	2	2	1	100	2
	dec.run_dec()	2	2	1	100	2
	inc.guard_inc()	2	2	1	100	2

	inc.run_inc()	2	2	1	100	2
	mini.guard_mini()	6	3	2	50	2
	mini.run_mini()	8	4	3	50	2
Totals		22	15	9		

Compilation time	18
Instrumentation time	190
Generation time	65
User time	243
Sys time	22

Table A.4: Evaluation table for the project MIO_multi_threaded with the second architecture.

Tool	Method under test	Total branches	Covered branches	Functions invoked	Covered branches (%
jmlcute	accept.guard_accept(Integer,Integer)	32	15	9	46,875
	accept.run_accept(Integer,Integer)	32	15	9	46,875
	accept.run()	32	15	9	46,875
	allow.guard_allow(Integer)	32	15	9	46,875
	allow.run_allow(Integer)	32	15	9	46,875
	allow.run()	32	15	9	46,875
	arrive.guard_arrive(Integer,Integer)	32	15	9	46,875
	arrive.run_arrive(Integer,Integer)	32	15	9	46,875
	arrive.run()	32	15	9	46,875
	CARD.guard_CARD(Integer,Integer)	32	15	9	46,875
	CARD.run_CARD(Integer,Integer)	32	15	9	46,875
	CARD.run()	32	15	9	46,875
	closeDoor.guard_closeDoor(Integer)	32	15	9	46,875
	closeDoor.run_closeDoor(Integer)	32	15	9	46,875
	closeDoor.run()	32	15	9	46,875
	DENIED.guard_DENIED(Integer)	32	15	9	46,875
	DENIED.run_DENIED(Integer)	32	15	9	46,875
	DENIED.run()	32	15	9	46,875
	disallow.guard_disallow(Integer)	32	15	9	46,875
	disallow.run_disallow(Integer)	32	15	9	46,875
	disallow.run()	32	15	9	46,875
	get_in_bus.guard_get_in_bus(Integer)	32	15	9	46,875
	get_in_bus.run_get_in_bus(Integer)	32	15	9	46,875
	get_in_bus.run()	32	15	9	46,875
	get_in_station.guard_get_in_station (Integer,Integer,Integer)	32	15	9	46,875
	get_in_station.run_get_in_station (Integer,Integer,Integer)	32	15	9	46,875
	get_in_station.run()	32	15	9	46,875
	get_out_bus.guard_get_out_bus(Integer)	32	15	9	46,875
	get_out_bus.run_get_out_bus(Integer)	32	15	9	46,875
	get_out_bus.run()	32	15	9	46,875

	get_out_station.guard_get_out_station (Integer,Integer)	32	15	9	46,875
	get_out_station.run_get_out_station (Integer,Integer)	32	15	9	46,875
	get_out_station.run()	32	15	9	46,875
	GRANTED.guard_GRANTED(Integer)	32	15	9	46,875
	GRANTED.run_GRANTED(Integer)	32	15	9	46,875
	GRANTED.run()	32	15	9	46,875
	GREEN_OFF.guard_GREEN_OFF (Integer)	32	15	9	46,875
	GREEN_OFF.run_GREEN_OFF(Integer)	32	15	9	46,875
	GREEN_OFF.run()	32	15	9	46,875
	greenOff.guard_greenOff(Integer)	32	15	9	46,875
	greenOff.run_greenOff(Integer)	32	15	9	46,875
	greenOff.run()	32	15	9	46,875
	leave.guard_leave(Integer)	32	15	9	46,875
	leave.run_leave(Integer)	32	15	9	46,875
	leave.run()	32	15	9	46,875
	openDoor.guard_openDoor(Integer)	32	15	9	46,875
	openDoor.run_openDoor(Integer)	32	15	9	46,875
	openDoor.run()	32	15	9	46,875
	PASS.guard_PASS(Integer)	32	15	9	46,875
	PASS.run_PASS(Integer)	32	15	9	46,875
	PASS.run()	32	15	9	46,875
	RED_OFF.guard_RED_OFF(Integer)	32	15	9	46,875
	RED_OFF.run_RED_OFF(Integer)	32	15	9	46,875
	RED_OFF.run()	32	15	9	46,875
	redOff.guard_redOff(Integer)	32	15	9	46,875
	redOff.run_redOff(Integer)	32	15	9	46,875
	redOff.run()	32	15	9	46,875
	reject.guard_reject(Integer,Integer)	32	15	9	46,875
	reject.run_reject(Integer,Integer)	32	15	9	46,875
	reject.run()	32	15	9	46,875
	UNBLOCK.guard_UNBLOCK(Integer)	32	15	9	46,875
	UNBLOCK.run_UNBLOCK(Integer)	32	15	9	46,875
	UNBLOCK.run()	32	15	9	46,875
Totals		2016	945	567	

Compilation time	309
Instrumentation time	601
Generation time	1003
User time	1624
Sys time	197

jcute	accept.guard_accept(Integer,Integer)	40	15	6	37,5
	accept.run_accept(Integer,Integer)	41,99475066	16	7	38,1
	accept.run()	66,00660066	28	10	42,42
	allow.guard_allow(Integer)	33,99915002	16	7	47,06

allow.run_allow(Integer)	36,01694915	17	8	47,2
allow.run()	78	39	13	50
arrive.guard_arrive(Integer,Integer)	10	10	4	100
arrive.run_arrive(Integer,Integer)	10	10	4	100
arrive.run()	16	13	5	81,25
CARD.guard_CARD(Integer,Integer)	14,00025455	11	5	78,57
CARD.run_CARD(Integer,Integer)	16	12	6	75
CARD.run()	90	45	14	50
closeDoor.guard_closeDoor(Integer)	10	10	4	100
closeDoor.run_closeDoor(Integer)	11,99563795	11	5	91,7
closeDoor.run()	17,99485861	14	6	77,8
DENIED.guard_DENIED(Integer)	10	10	4	100
DENIED.run_DENIED(Integer)	11,99563795	11	5	91,7
DENIED.run()	17,99485861	14	6	77,8
disallow.guard_disallow(Integer)	34,01360544	15	6	44,1
disallow.run_disallow(Integer)	36,03603604	16	7	44,4
disallow.run()	42,01680672	20	8	47,6
get_in_bus.guard_get_in_bus(Integer)	16	12	6	75
get_in_bus.run_get_in_bus(Integer)	18,00554017	13	7	72,2
get_in_bus.run()	23,988006	16	8	66,7
get_in_station.guard_get_in_station (Integer,Integer,Integer)	40	15	6	37,5
get_in_station.run_get_in_station (Integer,Integer,Integer)	41,99475066	16	7	38,1
get_in_station.run()	47,9616307	20	8	41,7
get_out_bus.guard_get_out_bus (Integer)	17,9910045	12	6	66,7
get_out_bus.run_get_out_bus(Integer)	20	13	7	65
get_out_bus.run()	26,01626016	16	8	61,5
get_out_station.guard_get_out_station (Integer,Integer)	17,9910045	12	6	66,7
get_out_station.run_get_out_station (Integer,Integer)	20	13	7	65
get_out_station.run()	26,01626016	16	8	61,5
GRANTED.guard_GRANTED(Integer)	10	10	4	100
GRANTED.run_GRANTED(Integer)	11,99563795	11	5	91,7
GRANTED.run()	17,99485861	14	6	77,8
GREEN_OFF.guard_GREEN_OFF (Integer)	10	10	4	100
GREEN_OFF.run_GREEN_OFF(Integer)	11,99563795	11	5	91,7
GREEN_OFF.run()	17,99485861	14	6	77,8
greenOff.guard_greenOff(Integer)	13,99491094	11	5	78,6
greenOff.run_greenOff(Integer)	16	12	6	75
greenOff.run()	21,9941349	15	7	68,2
leave.guard_leave(Integer)	16	11	5	68,75
leave.run_leave(Integer)	17,9910045	12	6	66,7
leave.run()	24	15	7	62,5

	openDoor.guard_openDoor(Integer)	18,00327332	11	5	61,1
	openDoor.run_openDoor(Integer)	20	12	6	60
	openDoor.run()	26,01626016	16	7	61,5
	PASS.guard_PASS(Integer)	12,06140351	11	5	91,2
	PASS.run_PASS(Integer)	12,06140351	11	5	91,2
	PASS.run()	17,99485861	14	6	77,8
	RED_OFF.guard_RED_OFF(Integer)	10	10	4	100
	RED_OFF.run_RED_OFF(Integer)	11,99563795	11	5	91,7
	RED_OFF.run()	17,99485861	14	6	77,8
	redOff.guard_redOff(Integer)	13,99491094	11	5	78,6
	redOff.run_redOff(Integer)	16	12	6	75
	redOff.run()	21,9941349	15	7	68,2
	reject.guard_reject(Integer,Integer)	40	17	6	42,5
	reject.run_reject(Integer,Integer)	41,95804196	18	7	42,9
	reject.run()	66,05922551	29	10	43,9
	UNBLOCK.guard_UNBLOCK(Integer)	10	10	4	100
	UNBLOCK.run_UNBLOCK(Integer)	11,99563795	11	5	91,7
	UNBLOCK.run()	17,99485861	14	6	77,8
Totals		1536,115152	920	395	

Compilation time	25
Instrumentation time	343
Generation time	1485
User time	1555
Sys time	204

Table A.5: Evaluation table for the project reversing_array with the second architecture.

Tool	Method under test	Total branches	Covered branches	Functions invoked	Covered branches (%)	Iteratior
jmlcute	final_evt.guard_final_evt()	30	15	8	50	3
	final_evt.run_final_evt()	30	15	8	50	3
	progress.guard_progress()	30	15	8	50	3
	progress.run_progress()	30	15	8	50	3
Totals		120	60	32		

Compilation time	230,6666667
Instrumentation time	454
Generation time	97,33333333
User time	704
Sys time	55

jcute	final_evt.guard_final_evt()	6	6	3	100	4
	final_evt.run_final_evt()	6	6	3	100	4
	progress.guard_progress()	6	6	3	100	4
	progress.run_progress()	6	6	3	100	4
Totals		24	24	12		

Compilation time	18
Instrumentation time	225
Generation time	101
User time	312
Sys time	25

Table A.6: Evaluation table for the project simple with the second architecture.

Tool	Method under test	Total branches	Covered branches	Functions invoked	Cover branc
jmlcute	ConjunctionOnCollection.guard (BSet,Integer)	20	10	5	50
	ConjunctionOnCollection.isSubsetGuard (BSet,Integer)	34,03141361	13	6	38,2
	ConjunctionOnCollection. guardCustomBSet (CustomBSet,Integer)	14	7	6	50
	ConjunctionOnCollection. guardPublicCustomBSet (PublicCustomBSet.Integer)	4	3	2	75
	ConjunctionOnIndirectCollection.guard (BSet,Integer)	13,98601399	6	4	42,9
	LinearArithmetic.add(int,int)	18,01801802	4	3	22,2
	LinearArithmetic.has(int)	10	8	4	80
	NonNull.guard(Object)	12,00480192	10	5	83,3
	NonNull.compleGuard(BSet)	13,99491094	11	6	78,6
	Simple.m1()	0	0	0	100
Totals		140,0351585	72	41	

Compilation time	209
Instrumentation time	386
Generation time	131
User time	635
Sys time	61

jcute	ConjunctionOnCollection.guard(BSet,Integer)	8	6	2	75
	ConjunctionOnCollection.isSubsetGuard (BSet,Integer)	22,00488998	9	3	40,9
	ConjunctionOnCollection. guardCustomBSet (CustomBSet,Integer)	6	3	2	50
	ConjunctionOnCollection. guardPublicCustomBSet (PublicCustomBSet.Integer)	2	2	1	100
	ConjunctionOnIndirectCollection.guard (BSet,Integer)	16	5	2	31,25
	LinearArithmetic.add(int,int)	2	2	1	100
	LinearArithmetic.has(int)	4	4	2	100

	NonNull.guard(Object)	4	4	2	100
	NonNull.compleGuard(BSet)	4	4	2	100
	Simple.m1()	0	0	0	100
Totals		68,00488998	39	17	

Compilation time	17
Instrumentation time	252
Generation time	130
User time	347
Sys time	36

Table A.7: Evaluation table for the project SM03_multi_threaded with the second architecture.

Tool	Method under test	Total branches	Covered branches	Functions invoked	Covered branches (%)	Iteration
jmlcute	ab.guard_ab()	24,01372213	14	7	58,3	3
	ab.run_ab()	24,01372213	14	7	58,3	3
	ab.run()	24,01372213	14	7	58,3	3
	ac.guard_ac()	24,01372213	14	7	58,3	3
	ac.run_ac()	24,01372213	14	7	58,3	3
	ac.run()	24,01372213	14	7	58,3	3
	ba.guard_ba()	24,01372213	14	7	58,3	3
	ba.run_ba()	24,01372213	14	7	58,3	3
	ba.run()	24,01372213	14	7	58,3	3
	ca.guard_ca()	24,01372213	14	7	58,3	3
	ca.run_ca()	24,01372213	14	7	58,3	3
	ca.run()	24,01372213	14	7	58,3	3
	cc.guard_cc()	24,01372213	14	7	58,3	3
	cc.run_cc()	24,01372213	14	7	58,3	3
	cc.run()	24,01372213	14	7	58,3	3
Totals		360,2058319	210	105		

Compilation time	239
Instrumentation time	488
Generation time	148
User time	771
Sys time	71

jcute	ab.guard_ab()	10	8	3	80	4
	ab.run_ab()	12	9	4	75	4
	ab.run()	32	22	7	68,75	4
	ac.guard_ac()	10	7	3	70	4
	ac.run_ac()	11,994003	8	4	66,7	4
	ac.run()	11,994003	8	4	66,7	4
	ba.guard_ba()	8	7	3	87,5	4
	ba.run_ba()	10	8	4	80	4
	ba.run()	10	8	4	80	4
	ca.guard_ca()	10	7	3	70	4

	ca.run_ca()	11,994003	8	4	66,7	4
	ca.run()	11,994003	8	4	66,7	4
	cc.guard_cc()	10	8	3	80	4
	cc.run_cc()	12	9	4	75	4
	cc.run()	12	9	4	75	4
Totals		183,976012	134	58		

Compilation time	19
Instrumentation time	250
Generation time	184
User time	379
Sys time	48

Table A.8: Evaluation table for the project SocialEventPlanner with the second architecture with jCUTE

Tool	Method under test	Total branches	Covered branches	Functions invoked	Covered branche
jcute	addDisjointness.guard(Integer,Integer)	24	12	6	50
	addDisjointness.run_add_disjointness (Integer,Integer)	26	13	7	50
	addDisjointness.run	32	16	8	50
	addtolist.guard(Integer,Integer)	10	10	4	100
	addtolist.run()	16	13	5	81,25
	addtolist.run_addtolist(Integer,Integer)	10	10	4	100
	comment_scontent_principal.run()	24	18	6	75
	comment_scontent_principal. guard_comment_scontent_principal (Integer,BSet,Integer)	10	10	4	100
	comment_scontent_principal. run_comment_scontent_principal (Integer,BSet,Integer)	10	10	4	100
	comment_scontent_secondary. guard_comment_scontent_secondary (Integer, BSet, Integer)	30,00054546	11	5	36,666
	comment_scontent_secondary. run_comment_scontent_secondary (Integer, BSet, Integer)	32	12	6	37,5
	comment_scontent_secondary.run()	45,97701149	20	8	43,5
	comment_wall.guard_comment_wall (Integer, Integer)	16	12	6	75
	comment_wall.run_comment_wall (Integer, Integer)	18,00005538	13	7	72,222
	comment_wall.run()	24,00024	16	8	66,666
	create_account.guard_create_account (Integer, Integer)	10	10	4	100
	create_account.run_create_account (Integer, Integer)	10	10	4	100

create_account.run()	16	13	5	81,25
create_list.guard_create_list(Integer, Integer)	50	21	8	42
create_list.run_create_list(Integer, Integer)	51,99962182	22	9	42,308
create_list.run()	58,00060321	25	10	43,103
create_social_event. guard_create_social_event (Integer)	10	10	4	100
create_social_event.run_create_social_event (Integer, Integer)	10	10	4	100
create_social_event.run()	16	13	5	81,25
delete_comment_wall. guard_delete_comment_wall (Integer, Integer)	10	10	4	100
delete_comment_wall. run_delete_comment_wall (Integer, Integer)	10	10	4	100
delete_comment_wall.run()	16	13	5	81,25
delete_list.guard_delete_list (Integer, Integer)	10	10	4	100
delete_list.run_delete_list (Integer, Integer)	10	10	4	100
delete_list.run()	16	13	5	81,25
delete_principal_scontent. guard_delete_principal_scontent (Integer, BSet, Integer)	36,00065456	11	5	30,555
delete_principal_scontent. run_delete_principal_scontent (Integer, BSet, Integer)	37,99993667	12	6	31,579
delete_principal_scontent.run()	51,99937601	20	8	38,462
delete_secondary_scontent. guard_delete_secondary_scontent (Integer, BSet, Integer)	30,00054546	11	5	36,666
delete_secondary_scontent. run_delete_secondary_scontent (Integer, BSet, Integer)	32	12	6	37,5
delete_secondary_scontent.run()	46,000276	20	8	43,478
edit_not_owned_scontent. guard_edit_not_owned_scontent (Integer, Integer, Integer)	30,00054546	11	5	36,666
edit_not_owned_scontent. run_edit_not_owned_scontent (Integer, Integer, Integer)	32	12	6	37,5
edit_not_owned_scontent.run()	37,999696	15	7	39,474
edit_owned_not_required_page. guard_edit_owned_not_required_page (Integer, Integer, Integer)	25,99981091	11	5	42,308

edit_owned_not_required_page. run_edit_owned_not_required_page (Integer, Integer, Integer)	28,00009333	12	6	42,857
edit_owned_not_required_page.run()	33,999728	15	7	44,118
edit_owned_not_required_planner. guard_edit_owned_not_required_planner (Integer, Integer, Integer)	30,00054546	11	5	36,666
edit_owned_not_required_planner. run_edit_owned_not_required_planner (Integer, Integer, Integer)	32	12	6	37,5
edit_owned_not_required_planner.run()	37,999696	15	7	39,474
edit_owned_required_page. guard_edit_owned_required_page (Integer, Integer, Integer)	25,99981091	11	5	42,308
edit_owned_required_page. run_edit_owned_required_page (Integer, Integer, Integer)	28,00009333	12	6	42,857
edit_owned_required_page.run()	33,999728	15	7	44,118
edit_owned_secondary_scontent. guard_edit_owned_secondary_scontent (Integer, Integer, Integer)	27,99979637	11	5	39,286
edit_owned_secondary_scontent. run_edit_owned_secondary_scontent (Integer, Integer, Integer)	30	12	6	40
edit_owned_secondary_scontent.run()	36,00057601	15	7	41,666
grant_edit_perm.guard_grant_edit_perm (Integer, Integer, Integer)	20	11	5	55
grant_edit_perm.run_grant_edit_perm (Integer, Integer, Integer)	22,00018333	12	6	54,545
grant_edit_perm.run()	28,000224	15	7	53,571
grant_populate.guard_grant_populate (Integer, Integer, Integer)	20	11	5	55
grant_populate.run_grant_populate (Integer, Integer, Integer)	22,00018333	12	6	54,545
grant_populate.run()	28,000224	15	7	53,571
grant_view_perm.guard_grant_view_perm (Integer, Integer, Integer)	18,00003273	11	5	61,111
grant_view_perm.run_grant_view_perm (Integer, Integer, Integer)	20	12	6	60
grant_view_perm.run()	26,00013867	15	7	57,692
hide.guard_hide(Integer, Integer)	18,00003273	11	5	61,111
hide.run_hide(Integer, Integer)	20	12	6	60
hide.run()	26,00013867	15	7	57,692
hide_comment_wall. guard_hide_comment_wall (Integer, Integer)	20	12	6	60

hide_comment_wall. run_hide_comment_wall (Integer, Integer)	22,00033847	13	7	59,09
hide_comment_wall.run()	27,99993	16	8	57,143
make_visible.guard_make_visible (Integer, Integer)	18,00003273	11	5	61,111
make_visible.run_make_visible (Integer, Integer)	20	12	6	60
make_visible.run()	26,00013867	15	7	57,692
reply_with_decline. guard_reply_with_decline (Integer, Integer)	16	11	5	68,75
reply_with_decline. run_reply_with_decline (Integer, Integer)	18,00018	12	6	66,666
reply_with_decline.run()	24	15	7	62,5
reply_with_join.guard_reply_with_join (Integer, Integer)	16	11	5	68,75
reply_with_join.run_reply_with_join (Integer, Integer)	18,00018	12	6	66,666
reply_with_join.run()	24	15	7	62,5
reply_with_maybe .guard_reply_with_maybe (Integer, Integer)	16	11	5	68,75
reply_with_maybe.run_reply_with_maybe (Integer, Integer)	18,00018	12	6	66,666
reply_with_maybe.run()	24	15	7	62,5
sent_invite.guard_sent_invite (Integer, Integer, Integer)	18,00018	12	6	66,666
sent_invite.run_sent_invite (Integer, Integer, Integer)	20	13	7	65
sent_invite.run()	26,000195	16	8	61,538
transmit_nolist.guard_transmit_nolist (BSet, Integer, Integer)	32	15	6	46,875
transmit_nolist.run_transmit_nolist (BSet, Integer, Integer)	33,9998725	16	7	47,059
transmit_nolist.run()	48	24	9	50
transmit_tolist.guard_transmit_tolist (BSet, Integer, Integer, Integer)	37,999696	15	6	39,474
transmit_tolist.run_transmit_tolist (BSet, Integer, Integer, Integer)	40	16	7	40
transmit_tolist.run()	54,00054001	24	9	44,444
transmit_tolist_restricted. guard_transmit_tolist_restricted (BSet, Integer, Integer, Integer, Integer)	43,99988267	15	6	34,091

	transmit_tolist_restricted. run_transmit_tolist_restricted (BSet, Integer, Integer, Integer, Integer)	45,99948251	16	7	34,783
	transmit_tolist_restricted.run()	60	24	9	40
	upload_principal.guard_upload_principal (Integer, Integer)	18,00003273	11	5	61,111
	upload_principal.run_upload_principal (Integer, Integer)	20	12	6	60
	upload_principal.run()	26,00013867	15	7	57,692
	upload_principal_content_planner. guard_upload_principal_content_planner (Integer, Integer, Integer)	24,00017455	11	5	45,833
	upload_principal_content_planner. run_upload_principal_content_planner (Integer, Integer, Integer)	25,99991333	12	6	46,154
	upload_principal_content_planner.run()	32	15	7	46,875
	upload_secondary.guard_upload_secondary (Integer, Integer, Integer)	18,00003273	11	5	61,111
	upload_secondary.run_upload_secondary (Integer, Integer, Integer)	20	12	6	60
	upload_secondary.run()	26,00013867	15	7	57,692
	upload_secondary_content_planner. guard_upload_secondary_content_planner (Integer, Integer, Integer, Integer)	24,00017455	11	5	45,833
	upload_secondary_content_planner. run_upload_secondary_content_planner (Integer, Integer, Integer, Integer)	25,99991333	12	6	46,154
	upload_secondary_content_planner.run()	32	15	7	46,875
Totals		2663,98149	1375	619	

Compilation time	28
Instrumentation time	326
Generation time	2706
User time	2564
Sys time	346

Table A.9: Evaluation table for the project SocialEventPlanner_multi_threaded with the second architecture with JMLCUTE.

Tool	Method under test	Total branches	Covered branches	Functions invoked	Covered branches
jmlcute	addDisjointness.guard(Integer,Integer)	61,99460916	23	12	37,1
	addDisjointness.run_add_disjointness (Integer,Integer)	105,8823529	36	16	34
	addDisjointness.run()	69,94818653	27	14	38,6
	addtolist.guard(Integer,Integer)	33,98926655	19	9	55,9
	addtolist.run()	40	22	10	55

addtolist.run_addtolist (Integer,Integer)	33,98926655	19	9	55,9
comment_scontent_principal.run()	48	27	11	56,25
comment_scontent_principal. guard_comment_scontent_principal (Integer,BSet,Integer)	33,98926655	19	9	55,9
comment_scontent_principal. run_comment_scontent_principal (Integer,BSet,Integer)	33,98926655	19	9	55,9
comment_scontent_secondary. guard_comment_scontent_secondary (Integer, BSet, Integer)	76,12456747	22	11	28,9
comment_scontent_secondary. run_comment_scontent_secondary (Integer, BSet, Integer)	100	26	14	26
comment_scontent_secondary.run()	91,98813056	31	14	33,7
comment_wall.guard_comment_wall (Integer, Integer)	46	23	12	50
comment_wall.run_comment_wall (Integer, Integer)	125,9259259	51	21	40,5
comment_wall.run()	54	27	14	50
create_account.guard_create_account (Integer, Integer)	33,98926655	19	9	55,9
create_account.run_create_account (Integer, Integer)	33,98926655	19	9	55,9
create_account.run()	40	22	10	55
create_list.guard_create_list (Integer, Integer)	80	31	14	38,75
create_list.run_create_list (Integer, Integer)	113,9896373	44	18	38,6
create_list.run()	88,0952381	37	16	42
create_social_event. guard_create_social_event (Integer)	33,98926655	19	9	55,9
create_social_event. run_create_social_event (Integer, Integer)	33,98926655	19	9	55,9
create_social_event.run()	40	22	10	55
delete_comment_wall. guard_delete_comment_wall (Integer, Integer)	33,98926655	19	9	55,9
delete_comment_wall. run_delete_comment_wall (Integer, Integer)	33,98926655	19	9	55,9
delete_comment_wall.run()	40	22	10	55
delete_list.guard_delete_list (Integer, Integer)	33,98926655	19	9	55,9

delete_list.run_delete_list (Integer, Integer)	33,98926655	19	9	55,9
delete_list.run()	40	22	10	55
delete_principal_scontent. guard_delete_principal_scontent (Integer, BSet, Integer)	88	22	11	25
delete_principal_scontent. run_delete_principal_scontent (Integer, BSet, Integer)	104,0723982	23	11	22,1
delete_principal_scontent.run()	104,0268456	31	14	29,8
delete_secondary_scontent. guard_delete_secondary_scontent (Integer, BSet, Integer)	76,12456747	22	11	28,9
delete_secondary_scontent. run_delete_secondary_scontent (Integer, BSet, Integer)	95,83333333	23	11	24
delete_secondary_scontent.run()	91,98813056	31	14	33,7
edit_not_owned_scontent. guard_edit_not_owned_scontent (Integer, Integer, Integer)	76,12456747	22	11	28,9
edit_not_owned_scontent. run_edit_not_owned_scontent (Integer, Integer, Integer)	133,9869281	41	19	30,6
edit_not_owned_scontent.run()	83,87096774	26	13	31
edit_owned_not_required_page. guard_edit_owned_not_required_page (Integer, Integer, Integer)	67,90123457	22	11	32,4
edit_owned_not_required_page. run_edit_owned_not_required_page (Integer, Integer, Integer)	215,8730159	68	30	31,5
edit_owned_not_required_page.run()	76,02339181	26	13	34,2
edit_owned_not_required_planner. guard_edit_owned_not_required_planner (Integer, Integer, Integer)	76,12456747	22	11	28,9
edit_owned_not_required_planner. run_edit_owned_not_required_planner (Integer, Integer, Integer)	236,1111111	68	30	28,8
edit_owned_not_required_planner.run()	83,87096774	26	13	31
edit_owned_required_page. guard_edit_owned_required_page (Integer, Integer, Integer)	68,11145511	22	11	32,3
edit_owned_required_page. run_edit_owned_required_page (Integer, Integer, Integer)	217,9487179	68	30	31,2
edit_owned_required_page.run()	76,02339181	26	13	34,2

edit_owned_secondary_scontent. guard_edit_owned_secondary_scontent (Integer, Integer, Integer)	71,89542484	22	11	30,6
edit_owned_secondary_scontent. run_edit_owned_secondary_scontent (Integer, Integer, Integer)	195,7831325	65	27	33,2
edit_owned_secondary_scontent.run()	80	26	13	32,5
grant_edit_perm.guard_grant_edit_perm (Integer, Integer, Integer)	55,97964377	22	11	39,3
grant_edit_perm.run_grant_edit_perm (Integer, Integer, Integer)	89,88764045	32	15	35,6
grant_edit_perm.run()	64,03940887	26	13	40,6
grant_populate.guard_grant_populate (Integer, Integer, Integer)	55,97964377	22	11	39,3
grant_populate.run_grant_populate (Integer, Integer, Integer)	89,88764045	32	15	35,6
grant_populate.run()	64,03940887	26	13	40,6
grant_view_perm. guard_grant_view_perm (Integer, Integer, Integer)	52,00945626	22	11	42,3
grant_view_perm. run_grant_view_perm (Integer, Integer, Integer)	83,98950131	32	15	38,1
grant_view_perm.run()	60,04618938	26	13	43,3
hide.guard_hide(Integer, Integer)	52,00945626	22	11	42,3
hide.run_hide(Integer, Integer)	96,1038961	37	16	38,5
hide.run()	60,04618938	26	13	43,3
hide_comment_wall. guard_hide_comment_wall (Integer, Integer)	53,99061033	23	12	42,6
hide_comment_wall. run_hide_comment_wall (Integer, Integer)	94,05940594	38	17	40,4
hide_comment_wall.run()	62,06896552	27	14	43,5
make_visible.guard_make_visible (Integer, Integer)	52,00945626	22	11	42,3
make_visible.run_make_visible (Integer, Integer)	89,97429306	35	15	38,9
make_visible.run()	60,04618938	26	13	43,3
reply_with_decline. guard_reply_with_decline (Integer, Integer)	48,0349345	22	11	45,8
reply_with_decline. run_reply_with_decline (Integer, Integer)	92,00968523	38	17	41,3
reply_with_decline.run()	56,03448276	26	13	46,4

reply_with_join.guard_reply_with_join (Integer, Integer)	48,0349345	22	11	45,8
reply_with_join.run_reply_with_join (Integer, Integer)	92,00968523	38	17	41,3
reply_with_join.run()	56,03448276	26	13	46,4
reply_with_maybe. guard_reply_with_maybe (Integer, Integer)	48,0349345	22	11	45,8
reply_with_maybe. run_reply_with_maybe (Integer, Integer)	92,00968523	38	17	41,3
reply_with_maybe.run()	56,03448276	26	13	46,4
sent_invite.guard_sent_invite (Integer, Integer, Integer)	50	23	12	46
sent_invite.run_sent_invite (Integer, Integer, Integer)	100	38	20	38
sent_invite.run()	57,93991416	27	14	46,6
transmit_nolist. guard_transmit_nolist (BSet, Integer, Integer)	33,98926655	19	9	55,9
transmit_nolist.run_transmit_nolist (BSet, Integer, Integer)	33,98926655	19	9	55,9
transmit_nolist.run()	82,00455581	36	15	43,9
transmit_tolist.guard_transmit_tolist (BSet, Integer, Integer, Integer)	33,98926655	19	9	55,9
transmit_tolist.run_transmit_tolist (BSet, Integer, Integer, Integer)	33,98926655	19	9	55,9
transmit_tolist.run()	93,99477807	36	15	38,3
transmit_tolist_restricted. guard_transmit_tolist_restricted (BSet, Integer, Integer, Integer, Integer)	33,98926655	19	9	55,9
transmit_tolist_restricted. run_transmit_tolist_restricted (BSet, Integer, Integer, Integer, Integer)	33,98926655	19	9	55,9
transmit_tolist_restricted.run()	105,8823529	36	15	34
upload_principal. guard_upload_principal (Integer, Integer)	52,00945626	22	11	42,3
upload_principal. run_upload_principal (Integer, Integer)	115,942029	40	16	34,5
upload_principal.run()	60,04618938	26	13	43,3
upload_principal_content_planner. guard_upload_principal_content_planner (Integer, Integer, Integer)	63,95348837	22	11	34,4

	upload_principal_content_planner. run_upload_principal_content_planner (Integer, Integer, Integer)	145,9627329	47	20	32,2
	upload_principal_content_planner.run()	72,02216066	26	13	36,1
	upload_secondary. guard_upload_secondary (Integer, Integer, Integer)	52,00945626	22	11	42,3
	upload_secondary.run_upload_secondary (Integer, Integer, Integer)	1027,777778	37	16	3,6
	upload_secondary.run()	60,04618938	26	13	43,3
	upload_secondary_content_planner. guard_upload_secondary_content_planner (Integer, Integer, Integer, Integer)	63,95348837	22	11	34,4
	upload_secondary_content_planner. run_upload_secondary_content_planner (Integer, Integer, Integer, Integer)	145,9627329	47	20	32,2
	upload_secondary_content_planner.run()	72,02216066	26	13	36,1
Totals		8439,383361	2872	1348	

Compilation time	380,5
Instrumentation time	648
Generation time	1708,5
User time	2296
Sys time	293

Table A.10: Evaluation table for the project sorting_array with the second architecture.

Tool	Method under test	Total branches	Covered branches	Functions invoked	Covered branches (%)	Iterations
jmlcute	final_evt.guard_final_evt()	16	7	5	43,75	3
	final_evt.run_final_evt()	36,01108033	13	7	36,1	3
	prog1.guard_prog1()	12	6	4	50	3
	prog1.run_prog1()	12	6	4	50	3
	prog2.guard_prog2()	12	6	4	50	3
	prog2.run_prog2()	12	6	4	50	3
	progress.guard_progress()	16	9	5	56,25	3
	progress.run_progress()	43,95604396	16	8	36,4	3
Totals		159,9671243	69	41		

Compilation time	243,25
Instrumentation time	512,5
Generation time	93,75
User time	762,25
Sys time	62,75

jcute	final_evt.guard_final_evt()	6	3	2	50	2
	final_evt.run_final_evt()	8	4	3	50	2
	prog1.guard_prog1()	8	4	2	50	2

	prog1.run_prog1()	8	4	2	50	2
	prog2.guard_prog2()	8	4	2	50	2
	prog2.run_prog2()	8	4	2	50	2
	progress.guard_progress()	5,997001499	4	2	66,7	2
	progress.run_progress()	8	5	3	62,5	2
Totals		59,9970015	32	18		

Compilation time	21
Instrumentation time	239
Generation time	84
User time	309
Sys time	27

Table A.11: Evaluation table for the project sqrt_number with the second architecture.

Tool	Method under test	Total branches	Covered branches	Functions invoked	Covered branches (%)	Iterations
jmlcute	final_evt.guard_final_evt()	10	5	5	50	3
	final_evt.run_final_evt()	10	5	5	50	3
	progress.guard_progress()	10	5	5	50	3
	progress.run_progress()	10	5	5	50	3
Totals		40	20	20		

Compilation time	229,5
Instrumentation time	475
Generation time	61,5
User time	697
Sys time	49

jcute	final_evt.guard_final_evt()	2	2	1	100	2
	final_evt.run_final_evt()	2	2	1	100	2
	progress.guard_progress()	2	2	1	100	2
	progress.run_progress()	2	2	1	100	2
Totals		8	8	4		

Compilation time	21
Instrumentation time	263
Generation time	57
User time	316
Sys time	19

Appendix B

Appendix JML Syntax

This appendix is edited from the JML Reference Manual: Lexical Conventions 4.6 Tokens [1] and describes the complete syntax of JML.

Character strings that are Java reserved words are made into the token for that reserved word, instead of being made into an ident token. Within an annotation this also applies to jml-keywords. The details are given below.

```
ident ::= letter [ letter-or-digit ] ...
letter ::= _, $, a through z, or A through Z
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
letter-or-digit ::= letter | digit
```

Several strings of characters are recognized as keywords or reserved words in JML. These fall into three separate categories: Java keywords, JML predicate keywords (which start with a backslash), and JML keywords. Java keywords are truly reserved words, and are recognized in all contexts. The non-terminal java-reserved-word represents the reserved words in Java (as in the JDK version supported by the tool in question, hopefully the latest official release).

The jml-keywords are only recognized as keywords when they occur within an annotation, but outside of a spec-expression store-ref-list or constrained-list. JML predicate keywords are also only recognized within annotations, but they are recognized only inside spec-expressions, store-ref-lists, and constrained-lists.

There are options to the JML tools that extend the language in various ways. For example, when an option to parse the syntax for the Universe type system is used, the words listed in the non-terminal java-universe-reserved also act like reserved words in Java (and are thus recognized in all contexts). When an option to recognize the Universe system syntax in annotations is used, these words instead act as jml-keywords and are only recognized in annotations. However, even when no Universe options are used, pure is recognized as a keyword in annotations, since it is also a jml-keyword. (The Universe type system support in JML is experimental. Most likely the list of java-universe-reserved will be added to the list of jml-keywords eventually.)

However, even without the Universe option being on, the jml-universe-pkeyword syntax is recognized within JML annotations in the same way as JML predicate keywords are recognized. The details are given below.

```
keyword ::= java-reserved-word
        | jml-predicate-keyword | jml-keyword
```

[1] http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_4.html

```
java-reserved-word ::= abstract | assert
        | boolean | break | byte
        | case | catch | char
        | class | const | continue
        | default | do | double
        | else | extends | false
        | final | finally | float
        | for | goto | if
        | implements | import | instanceof
        | int | interface | long
        | native | new | null
        | package | private | protected
        | public | return | short
        | static | strictfp | super
        | switch | synchronized | this
        | throw | throws | transient
        | true | try | void
        | volatile | while
        | java-universe-reserved  // When the Universe option is on
java-universe-reserved ::= peer | pure
        | readonly | rep
jml-predicate-keyword ::= \TYPE
        | \bigint | \bigint_math | \duration
        | \elemtype | \everything | \exists
        | \forall | \fresh
        | \into | \invariant_for | \is_initialized
        | \java_math | \lblneg | \lblpos
        | \lockset | \max | \min
        | \nonnullelements | \not_assigned
        | \not_modified | \not_specified
        | \nothing | \nowarn | \nowarn_op
        | \num_of | \old | \only_accessed
        | \only_assigned | \only_called
        | \only_captured | \pre
        | \product | \reach | \real
        | \result | \same | \safe_math
        | \space | \such_that | \sum
        | \typeof | \type | \warn_op
        | \warn | \working_space
        | jml-universe-pkeyword
jml-universe-pkeyword ::= \peer | \readonly | \rep
jml-keyword ::= abrupt_behavior | abrupt_behaviour
        | accessible | accessible_redundantly
        | also | assert_redundantly
        | assignable | assignable_redundantly
        | assume | assume_redundantly | axiom
        | behavior | behaviour
```

```
| breaks | breaks_redundantly
| callable | callable_redundantly
| captures | captures_redundantly
| choose | choose_if
| code | code_bigint_math |
| code_java_math | code_safe_math
| constraint | constraint_redundantly
| constructor | continues | continues_redundantly
| decreases | decreases_redundantly
| decreasing | decreasing_redundantly
| diverges | diverges_redundantly
| duration | duration_redundantly
| ensures | ensures_redundantly | example
| exceptional_behavior | exceptional_behaviour
| exceptional_example
| exsures | exsures_redundantly | extract
| field | forall
| for_example | ghost
| helper | hence_by | hence_by_redundantly
| implies_that | in | in_redundantly
| initializer | initially | instance
| invariant | invariant_redundantly
| loop_invariant | loop_invariant_redundantly
| maintaining | maintaining_redundantly
| maps | maps_redundantly
| measured_by | measured_by_redundantly
| method | model | model_program
| modifiable | modifiable_redundantly
| modifies | modifies_redundantly
| monitored | monitors_for | non_null
| normal_behavior | normal_behaviour
| normal_example | nowarn
| nullable | nullable_by_default
| old | or
| post | post_redundantly
| pre | pre_redundantly
| pure | readable
| refining
| represents | represents_redundantly
| requires | requires_redundantly
| returns | returns_redundantly
| set | signals | signals_only
| signals_only_redundantly | signals_redundantly
| spec_bigint_math | spec_java_math
| spec_protected | spec_public | spec_safe_math
| static_initializer | uninitialized | unreachable
| when | when_redundantly
```

```
        | working_space | working_space_redundantly
        | writable
        | jml-universe-keyword
jml-universe-keyword ::= peer | readonly | rep
```

The following describes the special symbols used in JML. The non-terminal java-special-symbol is the special symbols of Java, taken without change from Java.

```
special-symbol ::= java-special-symbol | jml-special-symbol
java-special-symbol ::= java-separator | java-operator
java-separator ::= ( | ) | { | } | '[' | ']' | ; | , | . | @
java-operator ::= = | < | > | ! | ~ | ? | :
        | == | <= | >= | != | && | '||' | ++ | --
        | + | - | * | / | & | '|' | ^ | % | << | >> | >>>
        | += | -= | *= | /= | &= | '|=' | ^= | %=
        | <<= | >>= | >>>=
jml-special-symbol ::= ==> | <== | <==> | <=!=>
        | -> | <- | <: | .. | '{|' | '|}'
        | <# | <#=
```

The non-terminal java-literal represents Java literals which are taken without change from Java.

```
java-literal ::= integer-literal
        | floating-point-literal | boolean-literal
        | character-literal | string-literal | null-literal

integer-literal ::= decimal-integer-literal
        | hex-integer-literal | octal-integer-literal
decimal-integer-literal ::= non-zero-digit [ digits ] [ integer-type-suffix ]
digits ::= digit [ digit ] ...
digit ::= 0 | non-zero-digit
non-zero-digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
integer-type-suffix ::= l | L
hex-integer-literal ::= hex-numeral  [ integer-type-suffix ]
hex-numeral ::= 0x hex-digit [ hex-digit ] ...
        | 0X hex-digit  [ hex-digit ] ...
hex-digit ::= digit | a | b | c | d | e | f
        | A | B | C | D | E | F
octal-integer-literal ::= octal-numeral  [ integer-type-suffix ]
octal-numeral ::= 0 octal-digit [ octal-digit ] ...
octal-digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

floating-point-literal ::= digits . [ digits ]
          [ exponent-part ] [ float-type-suffix ]
        | . digits [ exponent-part ] [ float-type-suffix ]
        | digits exponent-part [ float-type-suffix ]
        | digits [ exponent-part ] float-type-suffix
exponent-part ::= exponent-indicator signed-integer
exponent-indicator ::= e | E
```

```
signed-integer ::= [ sign ] digits
sign ::= + | -
float-type-suffix ::= f | F | d | D

boolean-literal ::= true | false

character-literal ::= ' single-character ' | ' escape-sequence '
single-character ::= any character except ', \, carriage return, or newline
escape-sequence ::= \b    // backspace
         | \t                  // tab
         | \n                  // newline
         | \r                  // carriage return
         | \'                  // single quote
         | \"                  // double quote
         | \\                  // backslash
         | octal-escape
         | unicode-escape
octal-escape ::= \ octal-digit [ octal-digit ]
         | \ zero-to-three octal-digit octal-digit
zero-to-three ::= 0 | 1 | 2 | 3
unicode-escape ::= \u hex-digit hex-digit hex-digit hex-digit

string-literal ::= " [ string-character ] ... "
string-character ::= escape-sequence
         | any character except ", \, carriage return, or newline

null-literal ::= null
```

An informal-description looks like (* some text *). It is used in predicates (see section 12.1 Predicates of the JML Reference Manual) and in store-ref expressions (see section 12.7 Store Refs of the JML Reference Manual) as an escape from formality. The exact syntax is given below.

```
informal-description ::= (* non-stars-close [ non-stars-close ] ... *)
non-stars-close ::= non-star
        | stars-non-close
stars-non-close ::= * [ * ] ... non-star-close
non-star-close ::= any character except ) or *
```