Bug-finding and Test Case Generation for Java Programs by Symbolic Execution

by

Willem Hendrik Karel Bester

Thesis presented in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science

at the University of Stellenbosch

Division of Computer Science
Department of Mathematical Sciences
University of Stellenbosch
Private Bag X1, Matieland 7602, South Africa

Supervisors:

Dr Cornelia P. Inggs   Prof. Willem C. Visser

December 2013

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Signature: Willem Hendrik Karel Bester
W. H. K. Bester

Date: 28 November 2013

Contents

List of Figures

List of Tables

Abstract

Bug-finding and Test Case Generation for Java Programs by Symbolic Execution

W. H. K. Bester

Division of Computer Science
Department of Mathematical Sciences
University of Stellenbosch
Private Bag X1, Matieland 7602, South Africa

Thesis: MSc (Computer Science)

December 2013

In this dissertation we present a software tool, Artemis, that symbolically executes Java virtual machine bytecode to find bugs and automatically generate test cases to trigger the bugs found. Symbolic execution is a technique of static software analysis that entails analysing code over symbolic inputs—essentially, classes of inputs—where each class is formulated as constraints over some input domain. The analysis then proceeds in a path-sensitive way adding the constraints resulting from a symbolic choice at a program branch to a path condition, and branching non-deterministically over the path condition. When a possible error state is reached, the path condition can be solved, and if soluble, value assignments retrieved to be used to generate explicit test cases in a unit testing framework. This last step enhances confidence that bugs are real, because testing is forced through normal language semantics, which could prevent certain states from being reached.

We illustrate and evaluate Artemis on a number of examples with known errors, as well as on a large, complex code base. A preliminary version of this work was successfully presented at the SAICSIT conference held on 1–3 October 2012, in Centurion, South Africa [9].

Uittreksel

Foutopsporing en Toetsgevalvoortbrenging vir Java-programme deur Simboliese Uitvoering

W. H. K. Bester

Afdeling Rekenaarwetenskap
Departement Wiskundige Wetenskappe
Universiteit van Stellenbosch
Privaatsak X1, Matieland 7602, Suid-Afrika

Tesis: MSc (Rekenaarwetenskap)

Desember 2013

In dié dissertasie bied ons 'n stuk sagtewaregereedskap, Artemis, aan wat biskode van die Java virtuele masjien simbolies uitvoer om foute op te spoor en toetsgevalle outomaties voort te bring om dié foute te ontketen. Simboliese uitvoering is 'n tegniek van statiese sagteware-analise wat behels dat kode oor simboliese toevoere—in wese, klasse van toevoer—geanaliseer word, waar elke klas geformuleer word as beperkinge oor 'n domein. Die analise volg dan 'n pad-sensitiewe benadering deur die domeinbeperkinge, wat volg uit 'n simboliese keuse by 'n programvertakking, tot 'n padvoorwaarde by te voeg en dan nie-deterministies vertakkings oor die padvoorwaarde te volg. Wanneer 'n moontlike fouttoestand bereik word, kan die padvoorwaarde opgelos word, en indien dit oplaasbaar is, kan waardetoekennings verkry word om eksplisiete toetsgevalle in 'n eenheidstoetsingsraamwerk te formuleer. Dié laaste stap verhoog vertroue dat die foute gevind werklik is, want toetsing word deur die normale semantiek van die taal geforseer, wat sekere toestande onbereikbaar maak.

Ons illustreer en evalueer Artemis met 'n aantal voorbeelde waar die foute bekend is, asook op 'n groot, komplekse versameling kode. 'n Voorlopige weergawe van dié werk is suksesvol by die SAICSIT-konferensie, wat van 1 tot 3 Oktober 2012 in Centurion, Suid-Afrika, gehou is, aangebied [9].

Acknowledgements

I wish to thank my supervisors, **Dr C. P. Inggs** and **Prof. W. C. Visser**; they have both showed me—no doubt, an infuriating student—great patience. The initial idea for this dissertation was Prof. Visser's, who is also responsible for the renewed interest in symbolic execution and some of the most exciting research in this area. His emphasis on and grasp of the "big picture", though exasperating from time to time, continues to be an inspiration. Dr Inggs, in particular, has been ever kind, and was always willing to help and listen, especially when my natural proclivities towards parsing knowledge LL(k) (where $k \to \infty$) threatened to derail all progress, but also when I was merely exhausted and feeling harried by my teaching load.

I also wish to thank **Prof. A. B. van der Merwe** for his willingness both to accept a masters student who had only vague ideas of what he wanted to accomplish and to cooperate with a non-resident primary adviser. As it turned out, Prof. van der Merwe is not listed as a member of a final triumvirate of supervisors, yet his initial suggestions permeate much of the original scaffolding of my practical work.

Finally, I must acknowledge my students at Stellenbosch University, especially the **Computer Science 214** (2010–11) and the **Computer Science 244** (2010–13) groups. They taught me anew about the difficulty of getting programs just right, the intransigence of programming tools, and the importance of understanding how first principles operate. Many of them, no doubt, suffered from some of my more outlandish excursions into software engineering practice, but I do hope they learnt as much as I did.

Dedications

TO MY PARENTS
who supported me uncritically and with love, both emotionally and financially, when I developed a quarter-life crisis and did not buy a bike, but went nuts and came back to university to start from scratch

TO SAM
who occupied my thoughts for far too long ... but so it goes, as Vonnegut said

TO LIDA
who read and wrote what I wrote and read, and was awake when everybody else was asleep

TO CECIL AND SUNETTE
who kept me sane through turbulence and tintinnabulation

TO SAKKIE
who taught me about hacking and life, equally

TO MY RATS EMMIE, SANDY, CREMORA, GRACE, AND SOOKIE
who taught me the value of continuing to move and making noise

TO THE MEMORY OF ALAN MATHISON TURING (1912–1954)
whom, though he was hounded into oblivion by the establishment, we owe so much:
AMOR ANIMI ARBITRIO SVMITVR, NON PONITVR

Chapter one

Introduction

As software use and applications have become increasingly pervasive in modern society, creating error-free software has become essential. This is motivated not only by issues of safety and infrastructural integrity in life-critical systems, but also by the cost of finding and fixing bugs in commercial enterprise [77].

Historically, two main avenues of writing error-free programs have been explored: On the one hand, the growing maturity of the software engineering discipline and related practices has resulted in cultural techniques of assisting programmers through different development strategies and testing methodologies; on the other hand, formal verification techniques have led to approaches that aim to automate (i) proving (at least partially) certain formal properties of program correctness or (ii) finding bugs.

In essence, bug-finding tools follow a middle-of-the-road approach: Program properties are not verified formally, yet formal methods are used to speed up and streamline the effort of bug discovery. Tools for finding bugs through program analysis follow either a **path-sensitive** or **path-insensitive** approach. In the latter case, techniques such as those based on **abstract interpretation** [4] aim to show the absence of errors. Path-sensitive approaches, however, typically focus on showing the presence of errors. In this endeavour, either a program is analysed entirely from its main entry point—for example, a `main` method—for **whole-program analysis** , or the publicly exposed methods of an Applications Progamming Interface (API) is explored **intraprocedurally** (one method at a time, without following method calls) or **interprocedurally** (one method at a time, but also following and exploring method calls).

In this dissertation, we present a tool Artemis that (i) symbolically executes Java bytecode to perform variably interprocedural analysis, (ii) uses constraint solving to determine feasible paths, and (iii) for feasible paths, generates test cases to show the presence of errors. Proceeding from Java bytecode, it neither assumes nor needs any specification except that implied by the API (namely, method signatures and return types), program assertions, and the assumption that the bytecode was produced by a valid and correct Java compiler that follows the Java language specification [37]. Using the API, assertions, and the documented compiler requirements, Artemis attempts to find the run-time exceptions that may be thrown by certain Java bytecode instructions, as specified by the Java virtual machine (JVM) specification [52]. In particular, Artemis is engineered to find and demonstrate violations of **safety properties**.

The errors indicated during program analysis may be spurious, either because the decision procedure cannot reason soundly over the input domain, or because there are certain environmental constraints to how object state can be set up. This implies that each error found must be considered in each possible context by a human, which seriously inhibits the usefulness of a bug-finding tool. To ameliorate the effort and to gain some extra assurance that the errors found are, in fact, real, Artemis generates at least one test case (formulated in a unit testing framework) for each error. These test cases are then run, and if an error could not be triggered, it is marked as potentially spurious.

Possibly the most important consequence of generating explicit test cases in a unit testing framework is that this makes Artemis instantly useful for regression testing. If an error is successfully triggered and then, subsequently, fixed in the code base, its test case(s) can still remain as part of a regression testing regime.

1.1 Method of Test Case Generation

1.1.1 Symbolic Execution

Artemis is based on **symbolic execution**. This technique was first proposed, motivated, and formally described in seminal papers by King [45, 46] and Clarke [18] as a practical approach to bug-finding, falling between the two extremes of formal program proving and more ad hoc program testing. It works in the absence of a formal specification and may be viewed as an enhanced testing technique. Instead of executing a program on the actual host architecture over a set of concrete sample inputs—generated randomly or following from other analyses—a program is executed over a set of classes of inputs, where each class is formulated as constraints over some input domain. These classes constitute "symbolic" input to the program, and importantly, the conjunction of particular constraints can be used to represent the program state.

Symbolic execution takes control-flow into account: In essence, it traverses a program's execution tree, which characterises the paths followed through the code during execution. For a program where control-flow is independent of its input, a single linear symbolic execution of the sequence of program statements suffices. But, for a program that contains branch statements over variables derived from its inputs, a **path condition** is stored that accumulates (that is, records the history of) the symbolic branch choices which led to a certain program statement (that is, a node in the execution tree).

Where the execution path relies only on concrete (non-symbolic) expressions—for any concrete branch guard q, either q or $\neg q$ is true, and its converse is false—a deterministic choice can be made, and the true branch is followed. However, if a branch guard q is symbolic, both the q and $\neg q$ branches must be explored. This is achieved by branching non-deterministically over the path condition pc—which is initialised to `true` when execution starts—and setting

$$pc_{\texttt{if}} \leftarrow pc \wedge q \tag{1.1.1}$$

for the `if` branch, and

$$pc_{\texttt{else}} \leftarrow pc \wedge \neg q \tag{1.1.2}$$

for the `else` branch. The paths specified by $pc_{\texttt{if}}$ and $pc_{\texttt{else}}$, respectively, are now explored recursively. Whenever a particular statement u is (i) reached by at least one non-deterministic branch, and (ii) it is known that the execution of u may cause a particular run-time exception unless a conjunction r of constraints is true (and therefore, $\neg r$ allows the exception), the constraint $s = pc_u \wedge \neg r$, where pc_u specifies the path whereby u was reached, is sent to a constraint solver. If s is deemed feasible by the constraint solver, the value assignments that make s true can be retrieved from the solver, and those values that correspond to inputs can be used to test whether the expected exception in u can be triggered for a set of concrete inputs.

1.1.2 Path-Sensitive Analysis

To prevent our analysis from attemping to traverse an infinite execution tre, which results from the symbolic execution of loops and recursion, statement revisitation must be limited, that is, bound in some way. In a path-sensitive approach, bounds are enforced on the branches through the execution, and not on individual program statements. Doing so allows the proper unrolling of loops, in particular, nested loops. If bounds were enforced on statements instead of on branches, some paths might be pruned prematurely, and thus, some possible error states not considered at all.

Note, however, that we tacitly assume methods to have no side-effects. In particular, if a specific sequence of (top-level) method calls is necessary to observe the object state leading to an error, our analysis will not deduce such a sequence.

1.1.3 Interprocedural Analysis

Interprocedural analysis follows method calls, and the call depth is limited by associating a counter d with each top-level method. This counter is initialised to some nonnegative value, and indicates how many lower levels of method calls are allowed. If a method m with $d = d_m$ calls a method m', then m' is executed with $d = d_m - 1$, and execution stops when $d < 1$. In the latter case, the return value of the call is taken to be an unconstrained symbolic value.

When $d = 0$, essentially we have intraprocedural analysis. In this case, all method calls from that method result in unconstrained, unknown symbolic values being used as return values. We also assume that execution of the method call did not result in changes to the global state.

```
1  public class ExtraMath {
2
3    public static int signum(int a) {
4      int ra;
5      if (a <= 0 || a >= 0)
6        ra = a / abs(a);
7      else
8        ra = 0;
9      return ra;
10   }
11
12   public static int abs(int b) {
13     int rb;
14     if (b < 0)
15       rb = -b;
16     else
17       rb = b;
18     return rb;
19   }
20
21 }
```

Figure 1.1: Naive Java implementations of the absolute value and signum functions.

1.2 A Motivating Example

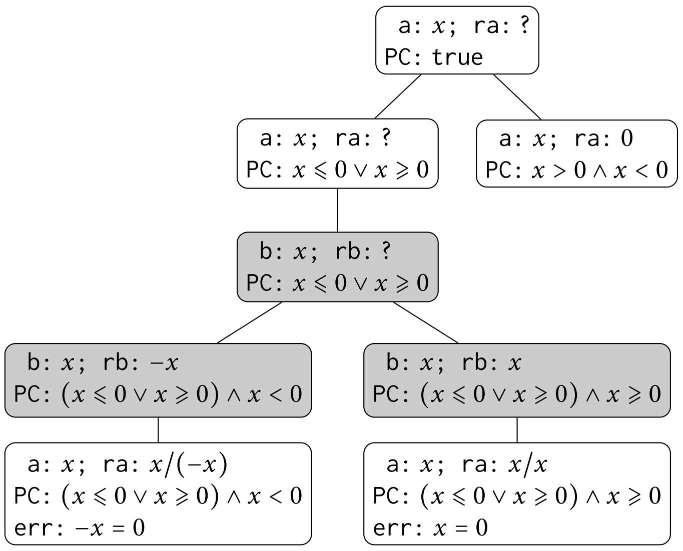
As an example of the usefulness of interprocedural analysis, consider the implementations of the absolute value and signum functions given in Figure 1.1. The class compiles without warning or error[*], but a problem lurks in the function `signum`: This function is a naive implementation, directly from a mathematical definition,

$$\text{signum}(a) = \begin{cases} a/|a| & \text{if } a < 0 \text{ or } a > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{1.2.1}$$

so that `signum` should return -1, 0, or 1 for a negative, zero, or positive argument, respectively. However, the `if` condition in line 5 was incorrectly entered, using the non-strict instead of the strict inequality relations. Therefore, control-flow always passes through line 6, and an `ArithmeticException` occurs for division-by-zero when `signum` is called with argument 0. Also, line 8 is effectively dead (unreachable) code.

In a traditional static control-flow analysis, the possibility of division-by-zero in line 6 will be reported. Using symbolic execution, we can determine (i) whether line 6 is reachable, and if so, (ii) which inputs lead to it. Figure 1.2 shows the symbolic execution tree for an

[*]Tested with Oracle Java JDK build 1.6.0_26 and Eclipse 3.6.1 builder, both on Linux.



a: x; ra: ?
PC: true
a: x; ra: ?
PC: $x \leqslant 0 \vee x \geqslant 0$
a: x; ra: 0
PC: $x > 0 \wedge x < 0$
b: x; rb: ?
PC: $x \leqslant 0 \vee x \geqslant 0$
b: x; rb: $-x$
PC: $(x \leqslant 0 \vee x \geqslant 0) \wedge x < 0$
b: x; rb: x
PC: $(x \leqslant 0 \vee x \geqslant 0) \wedge x \geqslant 0$
a: x; ra: $x/(-x)$
PC: $(x \leqslant 0 \vee x \geqslant 0) \wedge x < 0$
err: $-x = 0$
a: x; ra: x/x
PC: $(x \leqslant 0 \vee x \geqslant 0) \wedge x \geqslant 0$
err: $x = 0$

Figure 1.2: A symbolic execution tree for interprocedural analysis starting from `signum` in Figure 1.1. The vertices are the program states, and an edge denotes the program statement or method call leading to a particular state; the shaded states are those for the call to `abs`.

interprocedural analysis of the `signum` method in Figure 1.1. The analysis starts with the parameter `a` set to the symbolic integer value x, the path condition set to `true`, and the return value `ra` is undefined. The two children of the top vertex result from the non-deterministic branch over symbolic values in line 5; therefore, the path condition of the one is the negation of the other. The analysis is interprocedural, so the call to the method `abs` in line 6 is followed; these states are shaded in Figure 1.2. Note that the symbolic integer x is passed as argument to parameter `b` of `abs`, and also that the respective choices for the branch in line 14 are conjuncted with the existing path condition.

The tree has three leaves, each corresponding to a possible assignment of the return value `ra` in `signum`. The path to the rightmost leaf did not pass through line 6, and is therefore assumed to be safe and not considered any further. The other two, however, are sent to the constraint solver. In each case, a constraint that specifies division-by-zero and given as `err` in the figure, is conjuncted to the path condition. The constraint solver then determines the conjunction

$$(x \leqslant 0 \vee x \geqslant 0) \wedge (x < 0) \wedge (-x = 0) \tag{1.2.2}$$

for the leftmost leaf to be unsatisfiable, and the conjunction

$$(x \leqslant 0 \vee x \geqslant 0) \wedge (x \geqslant 0) \wedge (x = 0) \tag{1.2.3}$$

for the remaining leaf to be satisfiable for $x = 0$. Therefore, in the former case, the initial error indication in line 6 is assumed to be spurious, and no test case is generated; in the latter case, a test case is generated with the parameter `a` of `signum` passed a value of 0.

```
1  @Test public testSignum() {
2    try {
3      ExtraMath.signum(0);
4      // accounting: no exception
5    } catch (ArithmeticException e) {
6      // accounting: expected exception
7    } catch (Exception e) {
8      // accounting: other exception
9    }
10 }
```

Figure 1.3: Generated JUnit test case for the `signum` method of Figure 1.1.

The skeleton of a generated JUnit (version 4) test case for this example is shown in Figure 1.3. The commented lines could be changed, depending on why the analysis is performed. For example, as part of regression testing, reaching line 4 (that is, no exception thrown) is viewed as success, whereas for bug-finding, reaching line 6 (that is, catching the exception indicated by the analysis) is viewed as success, showing the presence of a bug.

1.3 Artemis: A Bug Finder and Test Case Generator

Artemis analyses Java bytecode directly, that is, without access to the source code. It can perform whole-program analysis, starting from a designated entry point, for example, a `main` method, or it can test interfaces, analysing all publicly exposed methods in a set of classes. Analysis proceeds as follows:

1. Java bytecode is converted, via the Soot Java optimisation framework [82], to a format amenable to symbolic execution, which is then executed over symbolic inputs by Artemis's symbolic execution engine.

2. If a particular path in the execution tree leads to a possible error over the symbolic inputs, the current state of that path, containing the path condition and other information to restrict the error domain, is sent to a constraint solver.

3. If the constraint solver finds a solution for the path state it was sent in the previous step, test cases are generated, where the original symbolic inputs are replaced with the solutions provided by the solver, and dumped to JUnit source files.

4. All the JUnit source files are collected, compiled, run, and only those test cases that manage to trigger the expected exception are marked as real errors.

1.3.1 Bytecode Execution

Artemis uses the Soot framework [82] to transform the Java bytecode representation found in Java `class` files into Jimple, a typed three-address Intermediate Representation (IR) available in Soot. Jimple consists of 15 statement and 45 expression types, which essentially replace intermediate results on the JVM stack with expressions stored in additional local variables.

1.3.1.1 Symbolic Expression Hierarchy

Artemis defines its own class hierarchy to model and simplify symbolic expressions, and the Jimple IR produced by Soot is translated into this hierarchy before symbolic execution by the Artemis engine. The following is a list of abstract classes that are extended to implementations for the given kinds of expressions:

- `BinaryExpression` for binary arithmetic, comparison, and bitwise shift and logic operations;
- `ConcreteValue` for concrete (that is, non-symbolic) values of primitive Java types, and `null` for reference types;
- `Reference` for object and array base references;
- `ReferenceExpression` for object member and array element expressions;
- `UnaryExpression` for unary arithmetic and logic operations, as well as array length expression and numeric cast operations; and
- `UnknownValue` for unknown (that is, symbolic) primitive input values.

The operands for any particular operation (modelled by a concrete class from this hierarchy) are, in turn, symbolic expressions, so that any compound expression is represented as a tree of classes deriving from `SymbolicExpression`. Symbolic expressions can be simplified, and such simplifications are propagated up the hierarchy so that the simplest possible expressions, with respect to the unknown symbolic inputs, can be presented to the constraint solver.

1.3.1.2 State and Branches

During symbolic execution, Artemis explores—from the designated entry point—program paths, treating the fifteen Jimple statement types case-by-case. A state object is associated with each program path; this state object, initially empty, stores current expressions for (i) local variables, (ii) field values, (iii) array entries, (iv) method parameters, (v) the call depth, (vi) the path condition, and (vii) branch counters. As symbolic execution proceeds, the state object is continually updated to reflect the current values and expressions for variables and the previously-mentioned execution parameters.

As soon as non-deterministic branching is to take place, Artemis clones the state object so that each branch gets its own copy on which the analysis proceeds. The size of the set of state objects instantiated during a run therefore gives an indication of the number of paths explored.

Unlike previous work [79], Artemis does not associate a counter with each program statement, but rather only with each branching statement. This allows Artemis both to limit branching on concrete branch conditions and to execute nested loops properly. For example, in the latter case, a nested loop with symbolic branch conditions and a branch bound of n will properly execute the innermost loop body n^2 times, unless other conditions (for example, breaking out of the loop) forces early exit from a particular loop run.

Artemis follows exceptions that are explicitly thrown in the code under analysis, that is, those resulting from `throw` statements as opposed to those resulting from the execution semantics of Java bytecode. When an exception is thrown explicitly, Artemis checks the current context—namely, statement blocks in the current method—for a matching handler, that is, one that handles a superclass of the thrown exception (and where an exception is a superclass of itself). If a matching handler is found, execution proceeds with the first statement indicated by the handler; otherwise, execution of the current path stops, and the current state and exception is propagated to the caller, using the same mechanism as for a normal method return. As a unique symbolic executor is instantiated for each top-level method call, the set of return states for such method calls are examined for the presence of an unhandled exception. If such exceptions exist, the corresponding state objects are handed to the constraint solver.

For example, for the class `Div` in Figure 1.4, Artemis reasons that the exception explicitly thrown in line 5 can be uncaught in the methods `div` and `div2`, but not in `div1`. So, the state objects corresponding to these paths are handed to the constraint solver; in this case, they are soluble, for the `b` parameters equal to zero, and therefore, test cases are generated for both methods.

1.3.1.3 Run-Time Exception Handling

The following run-time exceptions can be handled:

- `ArithmeticException` on integer division by zero;
- `ArrayIndexOutOfBoundsException` on array element references;
- `NegativeArraySizeException` on (explicit) array instantiation; and
- `NullPointerException` on instance field and array element references, instance method calls, and array length queries.

Symbolic execution can trigger an exception in two different ways: (i) on concrete conditions, and (ii) on symbolic conditions. In the former case, the expected exception is marked

```java
public class Div {

  public static int div(int a, int b) {
    if (b == 0)
      throw new ArithmeticException();
    return a / b;
  }

  public static int div1(int a, int b) {
    try {
      return div(a, b);
    } catch (ArithmeticException e) {
      return 0;
    }
  }

  public static int div2(int a, int b) {
    return div(a, b);
  }

}
```

Figure 1.4: A class to illustrate how Artemis follows exceptions.

as an **error**, execution of that particular branch stops, and the path condition is handed to the constraint solver to determine whether the path is feasible. In the latter case, the expected exception is possible unless an additional conjunction r of constraints hold; so, the constraint $pc \wedge \neg r$, where pc is the path condition, is delivered to the constraint solver. However, the branch is allowed to continue with the path condition $pc \wedge r$.

Consider, for example, a (symbolic) reference a to an array and a (symbolic) index i into this array. Indexing into the array is safe for

$$\begin{aligned} r &= 0 \leqslant i \leqslant a.\mathtt{length} \\ &= i \geqslant 0 \wedge i < a.\mathtt{length}, \end{aligned} \tag{1.3.1}$$

so that a warning is sent for solving on the constraint

$$pc \wedge \neg r = pc \wedge (i < 0 \vee i \geqslant a.\mathtt{length}), \tag{1.3.2}$$

while execution proceeds on the assumption

$$pc \wedge r = pc \wedge i \geqslant 0 \wedge i < a.\mathtt{length}. \tag{1.3.3}$$

Only if Eq. (1.3.2) can be satisfied will a test case be generated for this particular possible exception. Similar to non-deterministic branching, the state object is cloned for possible errors on symbolic conditions. So, the constraints of Eqs. (1.3.2) and (1.3.3) exist in different state objects.

1.3.2 Constraint Solving

Artemis checks path feasibility and calls for constraint solutions via the Green solver interface [83]. For Green, constraints are specified as simple constraint expression trees, similar to the symbolic expression hierarchy defined by Artemis. A simple translator class in Artemis suffices to bring constraint expressions into the required form for Green.

Since Artemis's path conditions are symbolic expressions over JVM types, it needs a constraint solver that supports reasoning at least over the integral and real domains; reference types can be modelled by picking special values from the integral domain. We tested three constraint solvers, namely, CHOCO [41] separately, and then CVC3 [8] and Z3 [62] as backends to Green.

1.3.3 Test Case Generation

If the constraint solver has determined that a path condition is feasible, Artemis extracts the solutions, and generates targeted JUnit [2] test cases via the StringTemplate engine [64]. For interface testing this implies providing parameters for method calls. Primitive types are handled by using solutions from the constraint solver for parameters bound by constraints or generating random values over the domains of those parameters that are not.

References, that is, object instances for parameters or instance methods calls, are more involved, because object state must first be set up. This entails selecting a constructor, which Artemis accomplishes by choosing the constructor for the containing class with the fewest parameters. If more than one reference type is necessary for the creation of a particular test case, Artemis computes the transitive closure over the object dependencies, recurring over parameters and the object base of a method call, so that object instantiation statements in the source code are written in the correct order.

This strategy is not without problems, however. For symbolic execution runs where an object or class field is read before written in the same method, that field could have (1) a default zero or `null` value, (2) a value after direct assignment, if the field is non-private, (3) a value assigned during some other method call, or (4) a value assigned by some constructor. The connection between constructors and field values is, therefore, tenuous at best, and may or may not exist. In Artemis, the problem is mitigated to some extent by keeping the generated test cases in the same package hierarchy as the class under analysis, which is to say, all but the private fields are accessible and can be handled by direct assignment. For private fields, Java's reflection API can be abused to make such fields public. We do so as a last resort, but classes secured by the Java's security manager can still refuse to allow this.

1.3.4 Crash Testing

The generated test source files are compiled with the system Java compiler, via the interface provided by the standard Java library. The compiled classes are then run by the JUnit library's core. In the source, each test case can have three outcomes, each kind of outcome accounted for separately: (1) A test case can fail to trigger the expected exception, (2) it can trigger the expected exception, or (3) it can trigger some other exception. The first case is viewed as a failure of the analysis, the second case as successfully showing the presence of a bug, whereas the last case is treated as a qualified success, since it indicated a problem, but not the one that was postulated by the analysis.

1.3.5 Contributions

Our work contributes to the field of bug-finding in a number of ways. First, and perhaps most important, we externalise testing through the generation and running of test cases in a standard testing framework. Doing so forces the bug-discovery process through the normal semantics and access control mechanisms of the Java language, meaning that it is difficult to create artificial bug scenarios that leaves spurious bugs to be weeded out manually.

Second, we handle loop unrolling well: We apply statement revisitation bounds in a path-sensitive manner, implying that paths are not truncated prematurely, and more involved paths are not skipped over during bug discovery.

Third, we handle run-time exceptions that are thrown explicitly by `throw` statements in the code, as opposed to handling just those that originate directly in the JVM. This is important since run-time exceptions in Java are unchecked, that is, the Java compiler does not check that they are caught by any of the parents in the hierarchy of calls leading to a run-time exception.

Chapter two

Background and Literature

In his *passages from the life of a philosopher*, Charles Babbage wrote of a discussion with Ada Augusta, Countess of Lovelace, "only child of the Poet Byron". Lovelace had translated a paper by Gen. L. F. Menabrea,[*] *Notions sur la Machine Analytique de M Charles Babbage* ("Ideas on the Analytical Machine of Mr Charles Babbage"; own translation), and Babbage suggested she add some original notes:

> We discussed together the various illustrations that might be introduced: I suggested several, but the selection was entirely her own. So also was the algebraic working out of the different problems, except, indeed, that relating to the numbers of Bernoulli, which I had offered to do to save the Lady Lovelace trouble. This she sent back to me for an amendment, *having detected a grave mistake which I had made in the process.* [7, p. 136; emphasis added]

Some recent historians [19, 55] have since apostatised from the pop-cultural, almost hagiographic treatment of Lovelace, not only seriously questioning the exact nature of her contributions to Babbage's work, but also challenging the veracity of Babbage's recollections. That notwithstanding, what the quoted passage does show is that the spectre of errors in algorithmic formulations for mechanical computing—as opposed to computing by humans—have been present since the very beginning of the programming discipline, and was acknowledged by the inventor of the Analytical Engine himself.

In this chapter, therefore, we shall embark on a brief journey through the major ideas in software analysis and verification. We first present a concise survey of general strategies for preventing or locating software errors. Then, in the sequel, we expound upon those ideas particularly relevant to our dissertation, and in doing so, lay the theoretical foundation on which our results ultimately rests.

In what follows immediately, we attempt a fairly general taxonomy of software analysis and verification. Although the terminology has become mostly standardised through use, the classification of techniques does not in all cases result in a clear hierarchical, or even orthogonal, structure. We point to such problems where it seems relevant or prudent to do so.

[*]Luigi Frederico Menabrea (1809–1896) was an Italian military engineer who served as Prime Minister of Italy, and later, as Italian ambassador to London and Paris. Menabrea wrote up the lectures presented by Babbage in August 1840 to the Academy of Sciences in Turin.

2.1 An Overview of Software Verification and Analysis

That computer systems exhibit errors has been long known [30]. A 2002 report has estimated the annual cost of software errors in the USA as almost $60 billion[†] [77], while some anecdotal evidence from industry suggests software professionals spend more than half their time testing and debugging [32, 69].

It might even be an interesting philosophical exercise to consider why this is the case. One might conjecture it has to do with the chimeric nature of computer science, and by extension, computer programming. On the one hand, it is a mathematical discipline, amenable to the methods and results of mathematics, but also vulnerable to its flaws and problems; on the other hand, it is an engineering discipline—particularly in the guise of software engineering, "[t]he application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software" [12]—and therefore, equally vulnerable to flaws and ambiguities admitted under this definition.

Putting etiological musings aside, we content ourselves, rather, with an intuitive teleological reasoning, that from the perspective of industry, software verification and analysis are motivated by the desire to prevent loss of money, time, life, and limb[‡] … to which those in academia, by no stretch of the imagination, might add the significance and sheer novelty of the academic pursuit, that is, a casus belli of *l'art pour l'art*.

Specification and Design There can be no clear notions of verification and analysis without some notions of **specification** (that is, *what* software should do) and **design** (that is, *how* an implementation accomplishes what it should). In this background chapter, we are more concerned with specification, though design certainly influences the choice of strategies for verification and analysis. Indeed, the very decision of what constitutes a software error in a particular context depends on the design [50]. Also, since narrative specification tends to be ambiguous or imprecise [50], many strategies of analysis require formal specification to varying degrees.

Software Quality How well software adheres to a specification is but one dimension of software quality. Some qualities that may be desirable under a given set of circumstances are listed and defined informally in Table 2.1 [20, 50, 51]. The degree to which a particular characteristic may be formalised and the rigour with which it may be employed as a quantifying metric is highly contextual. Although the strategies surveyed here may impact on any number

[†] Where in North American use 1 billion is taken as 1 000 000 000, that is $60 000 000 000. This figure is roughly 0.5% of the 2002 gross domestic of the USA, or equivalently, the 2002 GDP of the Czech Republic in constant 2000 dollars [data retrieved from the World Bank].

[‡] And, one would like to add somewhat facetiously, "face". Hoare [38], writing during the Cold War and Space Race, and with a sense for the dramatic, gives as examples of errors for which the costs are "incalculable—a lost spacecraft, a collapsed building, a crashed aeroplane, or a world war."

Table 2.1: Notions of software quality.

Characteristic	Description
Correctness	How well software adheres to its specific requirements
Efficiency	How well software fulfils its purpose without wasting resources
Maintainability	The ease of changing or updating software
Portability	The ease of using software across multiple platforms
Readability	How easily code can be read and understood
Reliability	The frequency and criticality of software failure
Reusability	How easily software components can be used by other software systems
Robustness	How gracefully software errors are handled
Security	The degree to which failure can cause damage
Usability	The ease with which users learn and execute tasks with the software

of these characteristics, we view correctness, given a particular context and problem domain, as the main goal. This focus may appear to be a jade's trick; however, correctness is the one characteristic of software quality that has been extensively studied—its theoretic foundations and application of theory to address its problems—for the past six decades.

Verification and Validation Boehm [11] gives as the basic objectives of software verification and validation of software requirements and design the "[identification and resolution] of software problems and high-risk issues early in the software lifecycle." He goes on to define **verification** as "establish[ing] the *truth* of the correspondence between a software product and its specification", and **validation** as "establish[ing] the *fitness* or *worth* of a software product for its operational mission". Informally then, verification asks whether we are building the software right, and validation asks whether we are building the right software.

In this dissertation we focus on software verification. Emerson [30] remarks that when sometimes the term *verification* is used within the specific context of establishing correctness, **refutation** (or **falsification**) is used with respect to error detection. Here we use the more general shorthand, where *verification* refers to a two-sided process of determining whether a software system is correct or has errors.

Automation of Analysis Kurshan [49] traces computer-aided verification back to Turing, and ultimately, to Russell and Whitehead's *Principia Mathematica*. Whereas Russell and Whitehead laid the foundation for axiomatic reasoning in the *Principia* [86], it is Turing's seminal paper *On Computable Numbers* [81] that led to the development of automata theory. On this edifice, much of the current state of the art has been founded.

We must, however, be careful of what the terms "automation" or "computer-aided" implies, and more specifically, to what degree any analysis is fully automated or merely aided by computer. Historically, there have been two approaches: On the one hand, software tools such as theorem provers and debuggers may be used in more or less manual analysis[§] of computer code; on the other, tools such as static analysers and model checkers may aim to analyse code without human intervention to the furthest possible extent.

We consider both approaches in this chapter, but our focus eventually falls on those techniques that reduces human intervention to a minimum. Such techniques cannot, however, be a panacea for the ills inherent to the programming disciple. Since questions of correctness are undecidable in general [21], there exists no magical elixir that renders human intervention completely unnecessary or superfluous. Of necessity, therefore, our survey must include such notions as *approximation* and *soundness*. The best we can hope to do is to formalise our notions of approximation, to pinpoint sources of and adopt strategies for handling (or at least, qualifying) unsoundness, and in the process, to reduce the number of cases that require human intervention.

Classification of Analysis Strategies Finally, we have to consider a basic taxonomy in which to organise and contextualise the strategies precised here: They are semantics-based, where **semantics** are the "relationships of symbols or groups of symbols to their meanings in a given language" [40], or equivalently, for a program in a programming language, "a formal mathematical model of all possible behaviours of a computer system executing this program in interaction with any possible environment" [21].[¶]

Laski and Stanley [50] divide the current state of the art of software analysis techniques into three categories:

1. **program proving**, where correctness of a program is demonstrated by proving consistency with its specification;
2. **static analysis**, where potential or real errors are detected without executing a program or explaining program behaviour (or both); and
3. **dynamic analysis**, where strategies such as debugging and structural testing, or techniques such as dynamic slicing are employed on running programs, that is, processes.

The division between the first two categories is somewhat arbitrary in that the mathematical results and techniques of program proving in most cases lay the foundation for static analysis. According to the brief descriptions above, the former aims to show correctness, whereas the latter aims to discover errors. But, with some hindsight, we also postulate here that automation is key: Although program proving may employ software for theorem proving, it still requires

[§]The idea here being that either the analysis is conducted mainly "on paper" like a mathematical proof, with a piece of software mechanically, and hopefully, exhaustively exploring the cases that must be considered, or that the software tool being used requires constant attention in some kind of interactive mode of operation.

[¶]This often contrasted with **syntax**, the "structural or grammatical rules that define how the symbols in a language are to be combined to form words, phrases, expressions, and other allowable constructs" [40]. Syntax analysis is a hallmark of the syntax-directed translation techniques found in most, if not all, modern compilers.

human ingenuity for the discovery and construction of the relevant lemmas, whereas tools for static analysis proceed automatically from a specification or a program, in the latter case possibly relying on assertions supported by a programming language itself or written for an external checking tool.

2.1.1 Program Proving

The **verification problem** may be formulated as the determination of whether or not a program M adheres to a given specification h [30]. If M is formulated as a Turing machine, given a specification h, this reduces to the halting problem, which, in general, is undecidable.

Standard literature on the topic of program proofs normally trace its origins to McCarthy [57, 58], who is credited with an "early statement of direction" [60], and who explored the simple expression of recursive functions and presented a method called *recursion induction*. However, it is instructive to note that the problem of program correctness was considered to some extent by Goldstine and Von Neumann [36], as well as Turing [80]. The first two authors noted that proofs of program correctness could, in principle, follow from a programmer's description of stepwise changes to the state of the vector of program variables [29].

Turing delivered a paper at the 1949 inaugural conference on EDSAC—the computer built under the direction of Maurice V. Wilkes at Cambridge University—and started with the following concise and prescient question [80]: "How can one check a routine in the sense of making sure it is right?" Turing proceeds with a motivation by analogy before giving a proof of a program with two nested loops and considering a general proof strategy, similar to that given by Floyd almost two decades later. However, there is no evidence that Turing's paper influenced later researchers in the field [60].

The first workable methods for program proofs were given by Naur, and separately, by Floyd. Naur introduced what he called the method of *general snapshots*, which are expressions of static conditions that hold whenever execution reaches particular points in an algorithm. He realised that proofs for "data processing" required the relation of the transformation defined by the algorithm to "a description of the transformation in some other terms, usually of the static properties" of the transformation's result [63]. As motivation, Naur used the example problem of finding the maximum element in an array: He notes that for an array A of length N, the index r of the maximum element can be related to other indices by the expression $A[i] \leqslant A[r]$ for $1 \leqslant i \leqslant N$ (and one-based indexing). As such, the result of the algorithm is described simply as the static property of being greater than or equal to another element, but the formulation neither specifies the process by which to find the maximum element, nor does it provide any guarantees that the result exists at all.

2.1.1.1 The Flowchart Semantics of Floyd

It is, however, a seminal paper by Floyd in 1967 that formed the foundation for the more formal approaches propounded later by Hoare in 1969 and Dijkstra in 1976. Floyd [31] proposed associating **assertions** (in essence, invariants) over first-order predicate calculus with each program statement—which he called *commands*—so that reasoning about a program's correctness reduces to reasoning about individual statements. He considered the safety property of *partial correctness*, as well as the liveness property of *termination*, and thus, *total correctness* [30]. We now describe his work in some detail since it formed the framework not only for program proving but also static analyses of software, in particular the method of symbolic execution.

Floyd illustrated his approach over **flowcharts**, a directed graph where each vertex is labelled by a command, and the edges represent the possible control-flow between commands. The semantic specification of the flowchart language is then given as an **interpretation** I, a mapping of the edges on propositions of which the free variables may be the variables of the program given by the flowchart. An edge e is said to be *tagged* by its associated proposition $I(e)$.

For a particular vertex, the incident edges are classified into *entrances* and *exits*, namely, the edges that enter and leave this vertex, respectively. It is now possible to formulate for each vertex its *antecedent* (or what would later be called its *precondition*) and *consequent* (or what would be called its *postcondition*). For k entrances $a_1, a_2, \ldots, a_k$, a vertex has k antecedents $P_i = I(a_i)$, where $1 \leqslant i \leqslant k$. Similarly, for ℓ exits $b_1, b_2, \ldots, b_\ell$, it has ℓ consequents $Q_j = I(b_j)$, where $1 \leqslant j \leqslant \ell$. It may also be useful to collect the antecedents and consequents in a natural way into the vectors $\mathbf{P} = (P_1, P_2, \ldots, P_k)$ and $\mathbf{Q} = (Q_1, Q_2, \ldots, Q_\ell)$.

Now, **verification** of a command c under a particular interpretation is a proof that if control enters c at an entrance a_i with P_i true, then if c is left at all, control leaves at an exit b_j with Q_j true. That is, verification should ensure that if control enters a vertex on a true antecedent, then there exists a true consequent by which it is left, if it is left at all.

A **semantic definition** of a given set of command types is a rule that constructs a **verification condition** $V_c(\mathbf{P}; \mathbf{Q})$ on the antecedent and consequent vectors of c. It is constructed so that for any command, a proof of the verification condition is a verification, according to the definition above, of that command. That is, reformulated as a logical implication, for a selected entrance with a true tag, if the verification condition is satisfied, then the tag of the selected exit will be true as well.

Of particular importance for later work, is the concept of a **counterexample** to a given interpretation of a command: An assignment of values to free variables, together with an entrance, that falsifies the logical implication of the verification condition. If no counterexample exists to any command interpretation that satisfies its verification condition, that semantic definition is called **consistent**; and if a counterexample exists for each command interpretation that does not satisfy its verification condition, that definition is called **complete**. A semantic definition must always be consistent, but completeness, though preferable, is not always possible.

Floyd formulates the requirements for a satisfactory semantic definition as four axioms which can also be deduced from the assumptions of consistency and completeness.

Axioms 2.1.1. For a semantic definition to be satisfactory, the following requirements must be met:

1. If $V_c(\mathbf{P};\mathbf{Q})$ and $V_c(\mathbf{P}';\mathbf{Q}')$, then $V_c(\mathbf{P}\wedge\mathbf{P}';\mathbf{Q}\wedge\mathbf{Q}')$.
2. If $V_c(\mathbf{P};\mathbf{Q})$ and $V_c(\mathbf{P}';\mathbf{Q}')$, then $V_c(\mathbf{P}\vee\mathbf{P}';\mathbf{Q}\vee\mathbf{Q}')$.
3. If $V_c(\mathbf{P};\mathbf{Q})$ and $V_c(\mathbf{P}';\mathbf{Q}')$, then $V_c\big((\exists x)(\mathbf{P});(\exists x)(\mathbf{Q})\big)$.
4. If $V_c(\mathbf{P};\mathbf{Q})$, $\mathbf{R}\vdash\mathbf{P}$, and $\mathbf{Q}\vdash\mathbf{S}$, then $V_c(\mathbf{R};\mathbf{S})$.

In the given order, these axioms can be used (1) to combine separate proofs of certain properties, (2) for case analysis, (3) to assert that if a variable has a property P on entry, its (possibly altered) value will have have property Q on exit, and (4) to assert that for a verifiable antecedent and consequent, a stronger antecedent and weaker consequent are also verifiable.

As for the actual verification conditions, Floyd considers the following five flowchart command types, the verification conditions of which appear in Table 2.2:

1. an assignment operation $x \leftarrow f(x,\mathbf{y})$, where x is a variable and f is an expression that may contain x and the vector $\mathbf{y}$ of other variables;
2. a branch command over the condition ϕ, with antecedent P_1, and consequents Q_1 and Q_2;
3. a join of control, with antecedents P_1 and P_2, and consequent Q_1;
4. the start of the program; and
5. a halt of the program.

In particular, note that the verification conditions for the first three command types specify how consequents can be deduced from the antecedents. Floyd emphasises that these semantic definitions follow in a natural way and that they are consistent and complete if the underlying deductive is. According to London [53], the verification conditions may be considered conjectures that show program correctness with respect to the supplied assertions, whenever they are all proved.

Now, for an argument over the execution semantics for the whole program, it is necessary that the antecedents be propagated through each command. This is accomplished by defining a transformation $T_c(\mathbf{P})$ for each command c, given the antecedent $\mathbf{P}$, such that for any set of semantic definitions,

$$V_c(\mathbf{P};\mathbf{Q}) \equiv (T_1(\mathbf{P}) \Rightarrow Q_1)\wedge\cdots\wedge(T_\ell(\mathbf{P}) \Rightarrow Q_\ell) \tag{2.1.1}$$

Table 2.2: Flowchart statement types and associated verification conditions.

Command type c	**Notation**	**Verification condition**
Assignment	$V_c(P_1; Q_1)$	$(\exists x_0)(x = f(x_0, \mathbf{y}) \wedge R(x_0, \mathbf{y})) \vdash Q_1$, where P_1 has the form $R(x, \mathbf{y})$
Branch	$V_c(P_1; Q_1, Q_2)$	$(P_1 \wedge \phi \vdash Q_1) \wedge (P_1 \wedge \neg\phi \vdash Q_2)$
Join of control	$V_c(P_1, P_2; Q_1)$	$P_1 \vee P_2 \vdash Q_1$
Start	$V_c(Q_1)$	Identically true
Halt	$V_c(P_1)$	Identically true

Table 2.3: Flowchart statement types and associated transformations.

Command type	**Notation**	**Strongest verifiable consequent**
Assignment	$T_1(P_1)$	$(\exists x)(x = S^x_{x_0}(f) \wedge S^x_{x_0}(P))$, where $S^x_{x_0}$ indicates the substitution of x_0 for x in the argument
Branch	$T_1(P_1)$	$P_1 \wedge \phi$ and $T_2(P_1)$ is $P_1 \wedge \neg\phi$
Join of control	$T_1(P_1, P_2)$	$P_1 \vee P_2$
Start		T_1 is false, so that $V_c(Q_1)$ is identically true
Halt		The set of T_js and Q_js is empty, so that $V_c(P_1)$ is identically true

for any variable interpretation and where T_j is of the form $T_{j1}(P_1) \vee \cdots \vee T_{jk}(P_{jk})$, it must be possible to substitute $T_j(\mathbf{P})$ for Q_j without loss of verifiability. Floyd's transformations for the five command types are given in Table 2.3.

Given that no (closed) loop exists with all edges untagged and that all loop entrances are tagged, it is possible to extend a partially specified interpretation to a complete specification, either by hand, or by some kind of mechanical proof system. Floyd proposes semantic definitions be cast into the form $V_c(\mathbf{P}; \mathbf{Q}) \equiv (T_c(\mathbf{P}) \vdash \mathbf{Q})$. Then the **strongest verifiable consequent** $\mathbf{T}_c(\mathbf{P})$ can be defined such that (most) semantic definitions are of the form

$$\mathbf{V}_c(\mathbf{P}; \mathbf{Q}) \equiv (\mathbf{T}_c(\mathbf{P}) \vdash \mathbf{Q}), \tag{2.1.2}$$

which admits some useful properties.

Properties 2.1.2. The **strongest verifiable consequent** has the following properties:

1. If $\mathbf{P} \Rightarrow \mathbf{P}_1$, then $\mathbf{T}_c(\mathbf{P}) \Rightarrow \mathbf{T}_c(\mathbf{P}_1)$.

2. If an executed command c is entered on a_i with initial values $\mathbf{V}$ and exited on b_j with final values $\mathbf{W}$, then $\mathbf{T}_c(\mathbf{P}) \equiv \mathbf{Q}$, where P_α is defined to be false for $\alpha \neq i$, $\mathbf{X} = \mathbf{V}$ for $\alpha = i$, Q_β is defined to false for $\beta \neq j$, and $\mathbf{X} = \mathbf{W}$ for $\beta = j$.

3. The transformation distributes over conjunction, disjunction, and existential quantification, that is,

a) If $\mathbf{P} = \mathbf{P}_1 \wedge \mathbf{P}_2$, then $\mathbf{T}_c(\mathbf{P}) \equiv \mathbf{T}_c(\mathbf{P}_1) \wedge \mathbf{T}_v(\mathbf{P}_2)$;

b) If $\mathbf{P} = \mathbf{P}_1 \vee \mathbf{P}_2$, then $\mathbf{T}_c(\mathbf{P}) \equiv \mathbf{T}_c(\mathbf{P}_1) \vee \mathbf{T}_v(\mathbf{P}_2)$; and

c) If $\mathbf{P} = (\exists x)(\mathbf{P}_1)$, then $\mathbf{T}_c(\mathbf{P}) \equiv (\exists x)(\mathbf{T}_c(\mathbf{P}_1))$.

If a semantic definition has these properties, it satisfies Axioms 2.1.1.

2.1.1.2 The Formal Approaches of Hoare

Although Floyd illustrated his method on a small subset of the Algol language, his paper does not give a general strategy of formulating **semantic axiomatics**, that is, the definition of a programming language as a proof system. This fell to a 1969 paper [38] by Hoare, who also introduced the so-called **Hoare triple**: The notation $\{P\}\ S\ \{Q\}$ is read as "[i]f the assertion P is true before the initiation of a program [or statement] S, then the assertion Q will be true upon its completion."‖ According to Dijkstra's terminology [27], P is called the **precondition**, and Q is called the **postcondition**. According to Hoare, a program's intended function—or that of a program part—"can be specified by making general assertions about the values which the relevant variables will take *after* execution of the program" [38]. He reinterpreted Floyd's work in terms of the following:

1. Axiom of Assignment: $\{P_0\}\ x := f\ \{P\}$ is a theorem, where x is a variable, := is the assignment operator, f is an expression without side effects but possibly containing x, and P_0 is obtained from P by substituting f for all occurrences of x.

2. Rules of Consequence: If $\{P\}\ S\ \{Q\}$ and $Q \Rightarrow R$ are theorems, then $\{P\}\ S\ \{R\}$ is a theorem. Similarly, if $\{P\}\ S\ \{Q\}$ and $R \Rightarrow P$ are theorems, then $\{R\}\ S\ \{Q\}$ is a theorem.

3. Rule of Composition: If $\{P\}\ S\ \{Q\}$ and $\{Q\}\ T\ \{R\}$ are theorems, then $\{P\}\ S; T\ \{R\}$ is a theorem, where the semicolon indicates procedural composition.

4. Rule of Iteration: If $\{P \wedge B\}\ S\ \{P\}$ is a theorem, then $\{P\}$ **while** B **do** $S\ \{P \wedge \neg B\}$ is a theorem, where the pseudocode specifies repetition of S while B is true.

Hoare's description of the Axiom of Assignment is particularly insightful in that he expected assignment to be treated "backwards", that is, we would derive the precondition from the postcondition. In this, he follows Floyd and points to Dijkstra's rules for inference of the precondition from the statement and the postcondition. It is also instructive to note in the Rule of Iteration that P is effectively a **loop invariant**, yet Hoare never called it that. Also, although present in modern texts on logic for computer science [39], we can only speculate over Hoare's omission of the following.

‖We use the modern notation; in the paper, Hoare put the braces around the symbol for the program, that is, he wrote $P\{S\}Q$.

RULE OF CONDITION: If $\{P \wedge B\}\ S\ \{Q\}$ and $\{P \wedge \neg B\}\ T\ \{Q\}$ are theorems, then $\{P\}$ **if** B **then** S **else** T $\{Q\}$ is a theorem.

A Hoare triple specifies **partial correctness**: Informally, $\{P\}\ S\ \{Q\}$ states that if a program S is executed from a memory state initially satisfying P, and S terminates, then afterwards, the memory satisfies Q. Similarly, **soundness** means that if $\{P\}\ S\ \{Q\}$ can be proven, then starting from a memory state initially satisfying P and executing S will only terminate in a memory state satisfying Q.

2.1.1.3 Program Termination

Until here, we have silently glossed over the issue of program termination (or equivalently, finiteness of repetitions), which arises where programs have loops. Indeed, the notion of correctness used in the previous section was called "partial" exactly because termination was not specified. The informal definition of partial correctness can be extended to **total correctness** by including the requirement of termination.

Floyd [31] considered the problem and proposed the construction of termination proofs over **well-ordered sets**, that is, sets in which each nonempty subset has a least member, or equivalently, sets which contain no infinite decreasing sequences. He defined a W-function to be a function of the free variables in a program interpretation, where the values of the function are taken from a well-ordered set. By the introduction of a new variable δ, not otherwise used in the program, Floyd defined for a command c the verification condition

$$V_c(P \wedge \delta = \phi \wedge \phi \in W; Q \wedge \psi \prec \delta \wedge \psi \in W), \tag{2.1.3}$$

that must be satisfied for termination, where the entrance of c is tagged by the proposition P and the W-function ϕ, its exit is tagged by the proposition Q and the W-function ψ, and $\prec$ is the ordering relation of the well-ordered set W. The proof should show that if a program is entered with initial values satisfying the tag of the entrance, it must terminate.

Wirth [87] formulated the same idea in much simpler form: For a loop condition B and loop body S, postulate an integer function N that depends on certain variables of the program such that each execution of S decreases the value of N, and if B is satisfied, then $N \geqslant 0$. If this function N can be shown to exist, that particular loop must terminate.

Wirth also recognised the importance of *loop invariants*—that is, an assertion that holds independently of the number of previously executed repetitions—when he wrote [87], "[t]he lesson that every programmer should learn is that the explicit indication of the relevant invariant for each repetition represents the most valuable element in every program documentation." In addition to his rules of analytic program verification, corresponding roughly to Floyd's treatment of flowgraphs, Wirth also gave two rules of derivation, (i) for a **while**-**do** construct, and (ii) for a **repeat**-**until** construct. These follow from a linearisation of the execution flow in a given loop: By "cutting" the loop—Wirth advocates cutting *before* the loop condition

B—and postulating an hypothesis H at the cut, assertions can now be derived through the linearised sequence of loop body statements. Here the assertion P at the end of the linearised sequence must be such that either $P \Rightarrow H$ or $H \Rightarrow P$.

At first glance, the aforementioned seems reasonable, and more importantly, tractable. However, the pioneering authors and those that followed them recognised the question of program termination to be equivalent to the halting problem. Wirth, in particular, pointed out that discovering the invariants for looping program flow is nontrivial. The result is that any attempts to automate program proving must be approximate or still involve human intelligence for the general case.

2.1.1.4 The Predicate Transformation Semantics of Dijkstra

In a 1976 book [27], Dijkstra introduced what is probably the best-known semantics for program proving. He defined the **weakest precondition** $\mathrm{wp}(S, Q)$ corresponding to the postcondition Q of the statement S as a condition that characterises the set of all states such that execution of S from these states will certainly result in proper termination, leaving the system in a final state satisfying Q. Since termination is specified, the weakest precondition semantics specify total correctness. So, if the state immediately before executing S is not in $\mathrm{wp}(S, Q)$, the final state does not satisfy Q or the system may fail to terminate.

Arguably anticipating automation, Dijkstra also formulated the weakest precondition as a **predicate transformer**, that is, for a fixed mechanism S, a rule that produces $\mathrm{wp}(S, Q)$ whenever it is fed the predicate Q. To this purpose, a stronger predicate P that implies $\mathrm{wp}(S, Q)$ is often acceptable in practice.

Properties 2.1.3. The **weakest precondition** has the following properties:

1. LAW OF THE EXCLUDED MIRACLE: $\mathrm{wp}(S, \mathbf{false}) = \mathbf{false}$.

2. MONOTONICITY: If $Q \Rightarrow R$ for all states, we also have $\mathrm{wp}(S, Q) \Rightarrow \mathrm{wp}(S, R)$.

3. $\mathrm{wp}(S, Q) \wedge \mathrm{wp}(S, R) = \mathrm{wp}(S, Q \wedge R)$.

4. $\mathrm{wp}(S, Q) \vee \mathrm{wp}(S, R) \Rightarrow \mathrm{wp}(S, Q \vee R)$.

Then, in a similar vein to Hoare [38], the following definitions can be used to characterise the semantics of programming languages.

Definition 2.1.4. For program semantics of abort, assignment, procedural composition, conditions, and repetition, define the **weakest precondition** as, respectively:

1. $\mathrm{wp}(\text{“}\mathbf{skip}\text{”}, Q) = Q$.

2. $\mathrm{wp}(\text{“}\mathbf{abort}\text{”}, Q) = \mathbf{false}$.

3. $\text{wp}(\text{“}x := E\text{”}, Q) = Q_{E \to x}$.

4. $\text{wp}(\text{“}S; T\text{”}, Q) = \text{wp}(S, \text{wp}(T, Q))$.

5. $\text{wp}(\text{“}\textbf{if } E \textbf{ then } S \textbf{ else } T\text{”}, Q) = \big(E \Rightarrow \text{wp}(S, Q) \wedge \neg E \Rightarrow \text{wp}(T, Q)\big)$.

6. $\text{wp}(\text{“}\textbf{while } E \textbf{ do } S\text{”}, Q) = \exists k \geqslant 0 : H_k$, where $H_0 = \neg E \wedge Q$ and $H_{k+1} = H_0 \vee \text{wp}(S, H_k)$.

Note that items 3 to 5, for assignment, procedural composition, and conditions, respectively, are formulated as backwards predicate transformers, that is, we reason from the postcondition to the precondition. Also note that item 6 is an inductive definition, and therefore, calls for inductive proofs.

Emerson [30] names, in particular, the compositional nature of the Floyd–Hoare approach, that is, that program proofs can be constructed from proofs of subprograms, as an important advantage. He also notes that, unfortunately, it does not scale well to large programs: Technical details can be overwhelming to a human, and the ingenuity required to formulate appropriate assertions for loop invariants, in particular, may render the approach prohibitive.

2.1.2 Static Program Analysis

The methods of static analysis have been successful in various subdisciplines of computer science:

1. Algorithms for *flow analysis* [50, 70] have been particularly useful for code optimisation, and have found their way into the canon for compiler design [5].

2. *Model checking* [16] has been used for the verification of concurrent finite-state systems [30], for example, non-terminating system programs and protocols.

3. *Abstract interpretation* [22] has aimed to provide a unified lattice-theoretic model for static analysis, formalising the theory of semantic approximation independently of particular applications.

For static analysis, the code representation employed is a significant factor: Too high a level of abstraction, and the results of the analysis may not be useful; too low a level of abstraction, and the results may miss the forest for the trees. The latter, in particular, is not always a clear-cut case. When we reason about correctness or try to find errors, both the abstract problem domain as well as specific implementations of a solution may come under analysis.

For flow analysis, for example, we typically deal with a specific implementation, yet we do not usually care for a precise syntactic representation. The **parse tree** yielded by a compiler represents the syntactic structure of a particular program [5], including vertexes for the syntactic categories of the programming language. For analysing the semantics of a program, much of this information is not essential. A simpler code representation, in the form of an **abstract syntax tree**, usually contains sufficient information, and often serves as the starting point of analysis. It is of more manageable size than the parse tree, and yields readily to mechanical transformation to other intermediate representations such as control-flow graphs.

2.1.2.1 Data and Control Flow Analysis

Up to the early 1970s a number of techniques evolved for the compile-time analysis of code to prevent redundant computations and optimise object code generated by compilers. Kildall [44] presented and justified a single algorithmic framework for (i) constant propagation, (ii) constant subexpression elimination, (iii) elimination of redundant register load operations, and (iv) live expression analysis.

Whereas type analysis for procedural languages (as opposed to object-oriented languages) are insensitive to permutations of program statements, the analyses listed in the previous section are sensitive to the flow of control. As such, **control-flow graphs** (CFGs) are used to illuminate dependencies of data and control [5, 50]. A CFG is a directed graph, in which **basic blocks** of statements are associated with vertexes, and edges show the possible flow of control [5, 70]. Here, a basic block is one where control enters the block only through the first statement, and control leaves the block without halting or branching, except possibly at the last block statement. The CFG is also constructed to have a single entry vertex and a single exit vertex. Using the CFG, starting from the entry or exit vertexes, code transformations can now be made (i) to prevent recalculation of already computed values, (ii) to prevent calculation of values that are never used, (iii) to reorder statements that are independent for improving register allocation, and (iv) to apply algebraic laws to intermediate representations in hope of simplifying computation [5].

The mathematical foundations for data-flow analyses are lattice theory and results following from Tarski's Fixed-Point Theorem. Operating in monotone or the stronger distributive frameworks, *transfer functions* are used in iterative algorithms to compute the flow of data, both for forwards and backwards analysis, the former computing information on past behaviour, and the latter computing information on future behaviour.

2.1.2.2 Abstract Interpretation

Lattice theory also forms the basis of **abstract interpretation** [4], first attempted as a rigorous formulation of approximating program analyses by Patrick and Radhia Cousot in the late 1970s [22]. In essence, the technique entails using a "refined", abstract analysis to approximate the execution of a more basic analysis, which is, in the limiting case, a concrete program [76]. Abstract analysis is based on flow analysis as described in the previous section, but the use of a correctness relation admits the possibility of mechanical proofs, and during the analysis, precision is sacrificed for speed.

If the concrete execution of a program is described as a sequence $v_0 v_1 \ldots v_i \ldots$ of transitions between (concrete) states v_i from some set V, an abstract interpretation of the program follows from using the properties ℓ_i from a complete lattice L as problem-dependent models for the states v_i. A **correctness relation** relates the states and properties in such a way that (1) a property under-approximates a state, that is, the property approximates all values of the state and possibly some others, and (2) if a value is approximated by more than one abstract value, the most precise can be selected as single approximation in the analysis. The operations of the lattice are used to specify the conjunction and disjunction of property information, and where the most precise abstraction needs to be selected, fixed-point results from the lattice-theoretic framework are used. The analysis is not limited to a single level of abstraction, and indeed, the correctness relation can be used to establish higher levels of abstraction.

The addition of a Galois connection to what is otherwise a flow analysis, allows mechanical correctness proofs.

Definition 2.1.5 (Galois connection)**.** A **Galois connection** $\langle L, \alpha, \gamma, M \rangle$ between two lattices L and M, with the partial orders $\sqsubseteq_L$ and $\sqsubseteq_M$, respectively, consists of two monotone functions $\alpha : L \to M$ and $\gamma : M \to L$ such that (1) $\alpha \circ \gamma \sqsubseteq_M \lambda m.m$ and (2) $\gamma \circ \alpha \sqsupseteq_L \lambda \ell.\ell$.

The lattice L, here, is the more precise property space, and M is the more abstract; the functions α and γ are therefore called the **abstraction** and **concretisation** functions, respectively. In tandem with the correctness relation, the first restriction of Definition 2.1.5 says that concretisation does not lose precision, and the second that, while abstraction might lose precision, it does not lose correctness. The transfer functions of flow analysis can therefore be defined in a systematic way. In theory, a basic analysis and a Galois connection can render the abstract analysis by mechanical computation. When such computation is too hard, both concrete and abstract analyses can be used in addition to the Galois connection as a starting point for verification against the conditions imposed by the theory. Indeed, in some practical cases, reasoning about the concretisation function suffices.

Although the Cousots originally were interested in analysing flow graphs, abstract interpretation has found its way not only into the area of formal methods [21, 67], but also into analyses for functional languages [4]. In the latter case, particularly, the method has been applied to *strictness analysis*, to avoid closures during lazy initialisation through the identification of parameters that can be passed by value, and to *in-place update analysis*, to determine when objects can be garbage-collected.

2.1.2.3 Model Checking

Clarke [16] notes that model checking "did not arise in a[n] historical vacuum." The exigency of finding concurrency bugs—which can be hard to reproduce, and therefore, difficult to find by testing—motivated an automatic method for determining whether a particular state-transition structure is a model of a particular formula, that is, answering the question of whether a program adheres to its specification. Model checking is, therefore, an instance of the verification problem [30], and Clarke [16] provides the following description:

> Let M be a Kripke structure (i.e., [a] state-transition graph). Let f be a formula of temporal logic (i.e., the specification). Find all states s of M such that $M, s \models f$.

It is important to note that although reasoning can be about infinite behaviour, the system itself is finite.

As is apparent from Clarke's description, the use of **temporal logic** is of critical importance. Temporal logic is a kind of modal logic that deals with different modalities viewed over points of time. Although temporal logic was suspected to be relevant to reasoning about computer systems in the 1960s [66], it was Pnueli [65] who made the crucial suggestion of its use for reasoning about ongoing concurrent programs, which can be characterised as *reactive systems*. Such systems, for example, microprocessors, operating systems, and communications protocols, typically exhibit the non-deterministic, ideally non-terminating behaviour that is difficult to analyse in the Floyd–Hoare framework.

Different temporal logics exist. Pnueli, for example, used what is today known as Linear Temporal Logic (LTL) [39], which contains operators for the basic temporal modalities (i) *sometimes*, (ii) *always*, (iii) *next time*, and (iv) *until*. The other oft-used example is Computational Tree Logic (CTL) [17], which allows branching in time [39]. It provides the basic temporal modalities (i) *for all futures* and (ii) *for some futures*, followed by one of the basic modalities of LTL.

Emerson [30] gives the following advantages of model checking over other verification techniques:

1. There is no need to construct a correctness proof by hand.
2. It is faster in comparison with other rigorous methods.
3. When a specification is not satisfied, a model checker will produce counterexamples, illustrating why the specification does not hold.
4. Since partial specifications can be handled, model checking can be used while the design of a complex system is still incomplete.
5. The temporal logics employed easily express the properties used in reasoning about concurrent systems.

Unfortunately, model checkers suffer from one major problem, namely, state-space explosion, which is inherent to any formulation where non-deterministic descriptions must be handled mechanically in a deterministic way. Therefore, during the past two decades, research has produced a number of strategies to deal with this problem [30]:

1. *Symbolic model checking* addresses the problem of state-space explosion for systems with many concurrent parts by using a symbolic representation, based on *ordered binary decision diagrams* for the state-transition diagrams, instead of simpler data structures such as adjacency lists.

2. *Partial order reduction* allows model checking to deal with asynchronous systems by exploiting the independence of concurrently executing events.

3. *Compositional reasoning* considers when the conjunction of local properties of small parts of the system under analysis imply the overall specification.

4. Techniques based on abstract interpretation can reduce the complexity of dealing with data when reasoning about reactive systems.

5. *Symmetry reduction* exploits replicated components in state-transition graphs.

6. With *parametrised systems*, an invariant process, representing the behaviour of a family of processes, is used to check properties of all family members at once.

2.1.3 Dynamic Program Analysis

As the name suggests, **dynamic program analyses** entail running program implementations on actual, concrete inputs. Unless the input domains are small enough so that they can be explored exhaustively, dynamic analysis is, typically, testing for errors. Testing is typically divided into two strategies [50]:

1. **Black-box testing** follows from the specification of a problem, and does not require any knowledge of the actual implementation. The specification, here, can be as simple as using the signature of the routine(s) under test, and a formal specification of constraints over the input and associated output values, respectively, pre- and postconditions. "Successful" testing would reveal an error by showing how particular concrete input, that adheres to given preconditions, results in output that does not adhere to given postconditions.

2. During **white-box testing**, tests are based on knowledge about an actual implementation, rather than a program specification. Since this is a structural exploration of the program, specification typically entails coverage criteria, whether over statements and branches, or of data flow by *definition–use chains* and generalisations thereof.

Laski and Stanley [50] argue that black-box testing is "the only practical method", and that structural testing serves to measure the adequacy of black-box testing. It also has the advantage of being able to show, by counterexample, that a specification is flawed.

The two crucial parts of black-box testing is then (i) synthesising the test cases, that is, determining the appropriate test inputs that map to (ii) the expected output. Determining these conditions typically falls into the domain of static analysis. For evaluating the test results, two strategies are readily apparent: (i) Compare output against that of an existing, trusted implementation, or (ii) use a *test oracle* that can determine compliance with the postconditions, without necessarily being able to reproduce the results. For example, if an implementation of some algorithm that sorts arrays is under consideration, it is comparatively easy to write an oracle to test whether the sorted array contains the original elements in sorted order.

When test cases are synthesised on an ad hoc-basis, certain basic strategies are often used to determine input and expected output:

1. Test special values, such as identity and zero elements, from the input domains.
2. Test fixed points, that is, for a function f, use values x such that $f(x) = x$.
3. Partition input into classes that are relevant to the input domain, for example, zero, negative and positive members of some numerical domain.
4. Test what happens at the domain boundaries, especially when the domain is some finite ordered set.

In addition to testing, another kind of dynamic analysis, namely, **trace analysis**, is possible. Recording the history of program state during execution for a particular input, known as a *trace*, can be useful when dealing with undecidable problems during static analysis. From one perspective, debugging a program via the inclusion of logging statements or with an interactive debugger, where the run-time areas of the program can be inspected and execution suspended, is a classic trace analysis. The technique of *dynamic program slicing* [48] is a different example of trace analysis: For a particular execution of a program, the subset of program statements that affect values of interest is computed so that the space to be explored by analysis can be reduced. It has been applied, in particular, to find data flow anomalies that involve structured and pointer data types.

2.2 Symbolic Execution

In retrospect, when Wirth wrote that "[p]rogram verification employs the same principles as empirical testing", he neatly summarised one of the most important aspects of what was soon to be known as *symbolic execution*: "But instead of recording the individual values of variables in a trace table, we postulate generally valid ranges of values and relationships among variables after each statement" [87]. Indeed, in his seminal 1976 paper [46], King referred to symbolic execution as "a practical approach between [the] two extremes" of program proving and program testing, and a recent survey [67] called it a "program analysis technique which represents program inputs with symbolic values instead of concrete, initialised data and executes the program by manipulating program expressions involving the symbolic values." Although King introduced his EFFIGY symbolic execution system in a 1975 paper [45], and Clarke [18] independently developed an engine for the Fortran language, it is King's 1976 paper that, judging by references in subsequent papers, stood for long as the urtext of the technique.

Structural (white-box) testing aims to exercise specific parts of a program [50]. A typical structural testing strategy might aim for some kind of statement or branch coverage, so that the dependency of control-flow upon input values becomes a significant factor. Unfortunately, the combinatorial explosion in the number of possibilities that must be considered for typical input domains soon renders naive coverage strategies prohibitively expensive, computationally speaking, unless the domains are suitably restricted.

The main idea behind symbolic execution is deceptively simple: Instead of executing a program over concrete input values selected from some domain, execute a program symbolically, where the inputs are symbolic values that are "assigned" when the execution starts, and the results of subsequent computations are given as functions of the symbolic inputs. The initial symbolic values, therefore, stand for classes of inputs. Symbolic execution then proceeds as a systematic exploration of a program's *execution tree* (discussed in more detail later), which characterises the execution paths followed through the program, the idea being that the result from a single path may be equivalent to a large number of test cases that are isomorphic in some essential way. Each execution path is distinguished from others by a *path condition*, an accumulation of the symbolic choices made for that particular path at the branch points in the program.

2.2.1 Symbolic Representation of Input Values

Where typical execution semantics is concerned with (i) the representation of data objects that can be stored in variables, (ii) the effects of program statements during the manipulation of data objects, and (iii) control-flow over data objects, the computational definitions of operations for symbolic execution are extended to cater for symbolic operands and to produce symbolic expressions as results. Therefore, King views symbolic execution as a natural extension to normal execution in that normal execution can be seen as a special case of symbolic execution. Informally, this special case follows from a sufficiently tight restriction applied with the usual relational operators to a particular atomic symbolic value.

An important aspect of symbolic execution as envisioned by King is that neither the language syntax, nor individual statements are changed (besides allowing for symbolic operands and results) in preparation for or during symbolic execution. So, although not explicitly mentioned by King, any code representation—including but not limited to source text, intermediate representations like control-flow graphs, and low-level machine instructions—could, in principle, be symbolically executed.

2.2.2 The Execution Tree, Branch Points, and Path Condition

Since symbolic execution explores paths through a program, the dependency of a program's control-flow on particular inputs determine which symbolic values are relevant to the analysis. To provide the essence of control-flow, the **execution tree** of a program can be generated as follows: (i) Add a vertex for each program statement, and (ii) indicate a transition (of control-flow) between two statements by a directed edge between the vertexes involved. In particular, for any branch statement, edges for both a true and false evaluation of the branch condition must be present in the graph. To keep current execution state, current values of the following are associated with each vertex: (i) the vector of variable values (most likely in the form of symbolic expressions), (ii) the program counter, and (iii) the path condition. A **path** is then a (possibly infinite) sequence of states, starting from the root of the tree, following one edge for each vertex.

Note that this execution tree does not correspond to the control-flow graph of a program, wherein each particular statement is present in only one vertex. Rather, it serves to illustrate the possible paths through a program, similar to the trees used to "unwrap" recursion. In particular where loops are concerned—since they can be framed in terms of conditional and unconditional branches in a control-flow graph—a particular statement may be present multiple times in the execution tree.

If control-flow is independent of input values, a single symbolic execution suffices. On the other hand, if control-flow depends on input values, we must resort to case analysis. The purpose and structure of the **path condition** now becomes apparent: As accumulator of the properties to be satisfied by inputs so that a particular path is followed, it is a conjunction of boolean expressions over the symbolic inputs. The path condition is always initialised to true, and each branch point in the original program representation then contributes to it.

For the typical programming language, a branch condition is written over variable evaluations, relational operators, and logical connectives. Similar to Dijkstra's handling of the weakest precondition for a conditional statement, at any branch point, two expressions can now be formed for the current state pc of the path condition and the branch condition q:

$$pc \Rightarrow q, \text{ and} \tag{2.2.1}$$

$$pc \Rightarrow \neg q, \tag{2.2.2}$$

where Eq. (2.2.1) corresponds to assumptions on pc leading to the **if** branch (alternatively, a true evaluation of pc), and Eq. (2.2.2) corresponds to the **else** branch (alternatively, a false evaluation for pc).

When exactly one of Eqs. (2.2.1) and (2.2.2) is identically true, King calls it *non-forking* execution of the particular branch statement, that is, each normal execution where pc is satisfied will follow, every time, the same alternative as the symbolic execution. When neither of the equations is identically true, that is, where a suitable assignment of values could make either true, where King calls it a *forking* execution. In this case, each of the alternatives must be explored in the form of a parallel execution of the two possibilities. King emphasises the importance of realising that a statement being executed forking or non-forking is associated with a particular execution and not that statement.

The non-determinacy of the parallel execution following the condition q is handled by updating the path condition thus:

$$pc \leftarrow pc \wedge q, \text{ and} \tag{2.2.3}$$

$$pc \leftarrow pc \wedge \neg q. \tag{2.2.4}$$

Here, Eq. (2.2.3) is for following the **if** branch, and Eq. (2.2.4) for the **else** branch. For non-forking branches, the path condition is not updated since, essentially, there is only one path through the particular branching point.

As described by King, the path condition can never become false. This is since (i) the path condition is initialised to true, and (ii) the only operation is an assignment of the form

$$pc \leftarrow pc \wedge r, \tag{2.2.5}$$

where r is a boolean expression as described previously. But this assignment is allowed only in the case where $pc \wedge r$ is satisfiable, that is, $\neg(pc \Rightarrow \neg r)$, which in turn is satisfiable if and only if $pc \Rightarrow \neg r$ is not a theorem.

King points out two properties of the execution tree under these conditions:

1. For each leaf (corresponding to a finite execution path), there exists a model (that is, an assignment of non-symbolic values) that traces the path under normal execution.

2. Since any two paths share a common, unique forking vertex, and since the path conditions never become false, the path conditions associated with any two leafs are distinct.

2.2.3 Some Observations and Comparisons to Other Frameworks

King considered symbolic execution from the viewpoints of both program proving and program testing. For the former case, all paths through the program must be considered. Then, as briefly justified by King, Floyd's verification conditions can be generated by symbolic execution of the program paths, and a proof for that path is simply a proof that the path condition at the end of the path implies the assertion associated with the statement at the final vertex, that is,

$$pc \Rightarrow \text{value}(B), \tag{2.2.6}$$

for the final assertion B. This establishes symbolic execution as a forwards analysis by the computation of a **strongest postcondition**, which was first explored by De Bakker [26] as a forwards counterpart to Dijkstra's notion of a weakest precondition. Dijkstra essentially started at a postcondition, and computed the weakest precondition for which it holds when a particular statement is executed; De Bakker works from a precondition to establish the strongest postcondition that holds after the execution of a statement.

Any program that has a finite symbolic execution tree does not require Floyd's inductive arguments. Following from the symbolic execution tree properties given in the previous section, the final vertexes of the paths is an exhaustive exploration of program behaviour and the program proof is reduced to proofs of Eq. (2.2.6) for each final vertex. For programs with infinite trees, no exhaustive exploration, and therefore, no absolute correctness proof is possible.

King mentioned the possibility of constructing inductive proofs over the infinite parts of a symbolic execution tree. However, he also pointed out an important difference between manipulating predicates for program proofs and for program testing: For testing, the expressions requiring proof are syntactically and semantically determined by the programming language, whereas for program proving, the predicate semantics derives from the problem area. This lead him to the conjecture that, at least in the short term, symbolic execution might be more useful for testing, that is, for finding counterexamples in the sense of Floyd.

The general thread of ideas through Floyd, Hoare, Dijkstra, and Wirth, leading to successful analysis by symbolic execution, should be readily apparent:

1. Programming idioms (such as programming languages or flowcharts) implicitly or syntactically define semantics.

2. Rules and procedures may be constructed to transform and propagate assertions following from the semantics between statements, so doing characterising the essentials of program state.

3. Programs that do not contain iteration with undetermined loop bounds may be explored exhaustively. Proofs of correctness, and conversely, finding errors, then reduce to proofs for the final states. If the underlying logic for the input domain is sound, then so is the program analysis.

4. For programs with unbounded loops, some form of approximation is necessary. Proofs may be attempted over inductive arguments, but generally, this is difficult in practice. In mechanical attempts we should therefore note the level of approximation carefully, use suitable restrictions, and be aware of inherent unsoundness in the particular analysis.

2.3 Related Work and Existing Tools

After the work of Boyer [13] and King [46], the technique of using symbolic execution for debugging and test case generation has languished in obscurity for the better part of two decades. Păsăreanu and Visser [67] have surveyed recent trends in the field. Before the renewed flurry of activity since the early 2000s, the technique was, however, applied to different programming languages, for example, Ada [28], Pascal [42], and others [34].

Our own work focuses on automatic test case generation; the work of Korel [47] is an earlier example. Automatic test case generation has also been approached from methods other than symbolic execution. Csallner et al., for example, developed the tools JCrasher [23] (for generating random JUnit tests), Check'n'Crash [24] (refining JCrasher tests; also using a constraint solver as part of the analysis), and DSD [25] (extending Check'n'Crash by an existing tool for inferring program invariants).

Recent work on symbolic execution and test case generation has included EXE [15] and KLEE [14]. EXE has been used to find bugs and the inputs that trigger them for, amongst others, packet filtering libraries, Linux file systems, and a regular expression library. KLEE uses symbolic execution to generate tests with high coverage, and was used to test, among other things, the GNU Coreutils utility suite. Our own work mirrors that of KLEE in that we also use symbolic execution and off-the-shelf solvers to generate tests; but whereas KLEE uses the LLVM compiler infrastructure, we use the JVM. Unlike KLEE, we also do not concern ourselves with code coverage. Instead, this aspect was treated in a separate project [61], in which a tool was implemented that uses Artemis's symbolic execution engine and test case generation module to determine code coverage.

There exist others tools that illustrate a technique called **concolic execution** [35, 54], under which execution starts over random concrete inputs, but path constraints are collected along the executed path. These are subsequently used to derive new inputs that force the execution down different, hitherto unexplored, paths. Specifically, the negation of the branch concrete condition at a particular branch point can drive the test generation process towards the other path. PEX [78], for example, targets the .NET platform, and performs systematic program analysis through symbolic execution to determine inputs for parametrised unit tests; it learns program behaviour by observing execution traces, and then uses a constraint solver to generate new inputs to effect different program behaviour.

Testing of different data types, that is, in addition to primitive numeric types, has also been explored, for example, pointers and structured data [89] and strings in Java [74]. CUTE [73, 72] extends the approach of concolic execution to handling recursive data structures by separating (and simplifying) pointer constraints from numeric constraints.

As discussed in §2.1.2.2, the over-approximations of abstract interpretation can be used to verify properties of a concrete program. However, under-approximations based on abstract interpretation can also be useful for property falsification, and therefore, program testing [6, 84]. Xu et al. [88], for example, have used symbolic execution to test for buffer overflows by under-approximation by keeping only length information for the lists and buffers under analysis.

Khurshid et al. [43] described the integration of symbolic execution with a model checker—via program instrumentation and using a decision procedure—to test the correctness of concurrent programs that take input from unbounded domains with complex structure. This work targets Java programs and uses the Java PathFinder (JPF) model checker [1] to explore the symbolic execution tree.

A new framework, Symbolic PathFinder (SPF) [68], does not require code instrumentation, but uses a non-standard Java bytecode interpreter on top of JPF. This tool has as one of its main applications the generation of test inputs to obtain high path coverage. SPF defers to the JPF core for representing program state (heap and stack values, and threads). From our perspective, this has the advantage that SPF can explore different thread interleavings, whereas we cannot. SPF also supports polymorphism with respect to data types during analysis. We do not do so currently, but we hope to extend Artemis to handle polymorphism during the test case generation phase, as opposed to during symbolic execution itself. Finally, it appears that SPF suffers from similar scalability issues as Artemis.

Chapter three

Design and Implementation

King's implementation of a symbolic execution tool, effigy, was written with interactive analysis in mind [45, 46]. His examples of its use constituted correctness proofs for small programs. We went the opposite direction in that we were interested in counterexamples that illustrate incorrect behaviour by triggering errors. Our idea was to implement a standalone automatic test case generator for programs in the Java language. Errors found during analysis may be spurious, either when the incompleteness in the decision procedure results in our using its results unsoundly, or when the required error state is not allowed to be set up by the normal language semantics. For example, there might be non-linear constraints where the solver can only reason over linear constraints, or the error state might require a second assignment to a `final` field. So, we wanted to have a test harness able to run the generated test cases and only mark as real those errors that managed to trigger an exception during normal execution of a piece of code. Under these qualifications, we were willing to accept the unsoundness of the underlying decision procedures, because following the lead of Tomb et al. [79], we expected the test harness to act as a filter.

Probably the most important initial decision concerned the problem domain: Just which errors, that is, exceptions, did we expect to try and find? Then, from King's 1976 paper [46], we knew we needed mechanisms to cater for the following:

1. a *sensible code representation*, close enough to the execution semantics of the Java language so that we could generate test cases, but free enough so that we could perform the necessary manipulations on symbolic values;
2. a *symbolic execution engine*, able to traverse an execution tree implicitly or explicitly, with a facility for keeping program state (including the path condition) for the various paths;
3. a notion of *limiting the execution to finite execution trees*, since we definitely intended to analyse programs with loops;
4. a hierarchy for the *representation of symbolic expressions*, allowing us to manipulate and simplify expressions, so as to implement Java's execution semantics symbolically; and
5. a *decision procedure* and *constraint solver* to automate finding solutions to path conditions and error predicates.

In addition, we also needed:

1. a way to *generate tests*, allowing for the idioms inherent to the object-oriented paradigm, for example, to set up object state; and

2. the actual *test harness*, able to collect the test cases, run them, and then to report the results.

Before we get to principles behind our solutions to these questions (§3.2), we first address the issue of the problem domain (§3.1). Then we proceed with a detailed presentation of the tool Artemis's implementation (§3.3), after which we conclude with some comments on our experiences during implementation (§3.4).

3.1 The Problem Domain

As a symbolic execution engine, Artemis was designed to perform structural analysis, that is, to explore execution paths through the code under analysis. By the argument that it is easier, with less opportunity for errors, to write an interpreter for a low-level language than for a high-level language, we decided to perform symbolic execution over Java bytecode rather than over Java source code. Doing so has a number of advantages:

1. It was neither necessary to write or find a fully functional Java parser, nor to consider transformations from parse tree through abstract syntax tree to some form of flow-graph amenable to symbolic analysis. Java is not a small language,* and although it is documented well via publicly available standard specifications (for example, in [37]), an implementation from scratch would not have been trivial. Also, especially since the takeover of Sun Microsystems by Oracle, there has been much community discussion on and effort to update the language. So, even if we had used an existing parser, we would have had to be sure that it was kept up to date.

2. By contrast, the Java virtual machine (JVM) has seen few updates—currently, version 3 as opposed to work on version 8 of the Java language. Also, for backwards compatibility, changes to the JVM are additive, whereas compliance with different versions of the language would have required a great deal of effort to ensure that metaphors were not mixed.

3. Other languages—for example, Clojure (a dialect of Lisp), Python, Ruby, and Scala, to name only a few—compile to the JVM. So, if our backend for test case generation was well decoupled from the symbolic execution engine, we conjectured it should be relatively easy to extend our work to such languages.

*The most recent Java Language Specification occupies 640 pages. In contrast, the C language specification of 1989 is an appendix of a 200-page manual.

4. By assuming that the Java bytecode under analysis was produced with a correct and standards-compliant compiler, we are able to make certain assumptions about possible errors, restricting the number of errors we have to handle.

Since we intended to analyse "real world" code, we did not, for example, want to mandate the use of annotation tools to affix specifications to be tested—this would have been tantamount to functional testing anyway. This, and the selection of Java bytecode as base representation, effectively fixed the problem domain: We could hope to find those exceptions specified for instructions of the JVM [52], as well as those thrown via explicit statements in the source language.

3.2 Design and Implementation Principles

In a research project of this nature, one dares to say, there is always the classic dichotomy between science and engineering so prevalent in computer science: On the one hand, we wanted a tool that is useful, but on the other, we did not have all the answers up front, so we needed a flexible approach. Coming from a Unix background, initially there was the temptation to go the Unix way by writing a set of functionally specific tools, each performing a separate part of the analysis, that could then be chained together in a pipeline. However, we wanted platform independence. Since we focused on Java programs, the obvious solution seemed to be a Java-only implementation of the system. In the end, the "Java-only" adjective proved to be too optimistic for the available constraint-solving technology, but the rest of the components are pure Java, without native code.

We were interested in *design patterns*, so some effort was made to study and employ the available literature [33, 59]. For the range of the interconnecting components, and considering the autonomy we granted them in computing results, the *visitor pattern* was found particularly useful.

As if our extensibility requirements were not enough, Bloch [10] soon convinced us of the desirability of well-designed APIs through the use of interfaces. Also, with components interacting in nontrivial ways, from the outset we programmed with the *throw low, catch high* exception paradigm in mind: (i) Privately scoped methods enforces pre- and postconditions with `assert` statements, (ii) the rest accomplishes the same to a somewhat more limited extent with run-time exceptions, and (iii) these assertions and exceptions are written as "deep" as possible, allowing them to propagate up the call stack.

It was clear that we had to write the symbolic execution engine and expression hierarchy, as well as the test generator, ourselves. That left decisions of using existing libraries for (i) code representation, (ii) constraint solving, and (iii) the test harness.

3.2.1 A Brief Description of the Java Virtual Machine

The Java virtual machine specification [52] essentially describes a virtual machine that understands and can execute instructions (that is, bytecodes) of a particular binary format, the Java `class` file. Such a file contains sequences of bytecodes, a symbol table, and other ancillary information to enable, amongst others, dynamic dispatch and security checks.

The JVM is a 32-bit stack machine, that is, it is not register-based, but uses a stack for computation: Load and store instructions read and write variables to and from an **operand stack**, while most other instructions pop their operands and push their results onto the stack. Each operand stack sits inside a **stack frame**, which is the global state object for the method being executed, and therefore, also contains linking information and an array for local variables and parameter passing. The stack frames are, in turn, also pushed to and popped from another stack (allocated per thread) to handle (possibly recursive) sequences of method calls. A global heap memory area, shared between all threads, is available for dynamic allocation. (Native code is handled in separate stacks.)

The JVM operates on two variable types: *primitive types* and *reference types*. The primitive types are 8-bit, 16-bit, 32-bit and 64-bit signed integers, 16-bit unsigned integers for characters, 32-bit and 64-bit floating-point numbers (conforming to the IEEE 754 standard), a `boolean` type, and a `returnAddress` type for returning from calls. Support for the `boolean` type is limited and program statements are mostly compiled to instructions on integers. Widening and narrowing conversions are allowed between the numeric types.

The `reference` types are used for memory pointers to class types, array types, and interface types. The special `null` value is allowed and may be cast to any type. Memory addresses on the heap are not reified, so no pointer arithmetic is possible, easing the effort of garbage-collection.

Local variables and parameters share the same array in a particular stack frame, with the parameters (if any) preceding the local variables. They are accessible by index and always occupy at least 32 bits, even the narrower types; wider types occupy 64 bits. The specification does not mandate alignment, but does not prevent it either. For instance methods, the `this` reference is always passed at index 0 of the local variable array.

Considering the bytecode instructions provided, the JVM is biased towards execution and branching on integral types. For example, in addition to a general instruction for loading constants on the stack, separate bytecodes are provided for pushing each of the small integral constants $-1, 0, 1, \ldots, 5$; these constants may also be pushed by the general instruction. Presumably, when assuming small integral constants occur frequently in most programs, the duplication in functionality allows smaller `class` files and shorter decoding time by the JVM execution engine. The branching instructions also display this asymmetry: For integral types single instructions compare and branch; for others, a comparison is made and its result pushed onto the stack, where after branching occurs by comparison (with 0) with respect to

```
public class SquareRoot {

  public static double sqrt(double c) {
    double epsilon = 1e-15;
    double t = c;

    while (Math.abs(t - c/t) > epsilon*t)
      t = (c/t + t) / 2.0;

    return t;
  }

}
```

Figure 3.1: A Java implementation of Newton's method for calculating the square root of a real number.

the result on the stack.

3.2.2 Code Representation

In the second edition of the JVM specification [52], 205 bytecodes are reserved for use. To facilitate conversion into symbolic expressions, we wanted to streamline the number of cases handled separately.

Soot is a Java framework [82] for code optimisation, but also capable of general static analysis. It provides a number of intermediate representations (IRs), one of which, Jimple, is a typed three-address IR. Jimple consists of 15 statement and 45 expression types, which essentially replace intermediate results on the JVM stack with expressions stored in additional local variables.

Example 3.2.1. Figure 3.1 contains a Java method that calculates the square root of a real number, specified as a double-precision floating-point number, in an iterative implementation of Newton's method [71]. A mnemonic representation of the method's bytecode, produced by the `javap` command-line tool after compilation by a standard Java compiler, is given in Figure 3.2. The output displays a number of typical features:

1. The number before each instruction gives its offset in the bytecode vector of the class file, and instructions that transfer control non-sequentially use these as operands to specify targets (always given with respect to the start of the bytecode vector);

2. numbers prefaced by a hash symbol are indices into the constant pool—in Figure 3.2 the value in the constant pool to which a particular index points is given in a double-dash comment;

3. a statement mnemonic linked to a number by an underscore specifies a instruction where the instruction–operand combination has a unique bytecode encoding;

4. loop tests typically appear at the end of the loop body in bytecode—refer to the jump from index 7 to index 23;

5. conditional branching for floating-point and 64-integral values happen first by an instruction that pushes −1, 0, or 1 onto the operand stack depending on whether the first operand is less than, equal to, or greater than the second operand, and then branching with respect to a comparison to zero.

Figure 3.3 gives the Jimple IR produced by Soot from the bytecode in Figure 3.2; statement numbers are for illustrative purposes only. Note that statement 1 shows the association of a parameter with a local variable in the local variable array, and that variable names prefaced a dollar sign gives virtual variables that stores the intermediary results on the operand stack. This Jimple code defines the control-flow graph also given in Figure 3.3 ■

The regularity of Jimple allowed us to use a more modest converter to symbolic expressions than would otherwise have been the case. In addition, its use of intermediary variables provided a more abstract view of memory as name–value pairs rather than the indexed vector-based design for variables and parameters in the JVM stack frame. In turn, this memory abstraction more naturally admitted storage in hashed structures, which turned out to be helpful for associating different value representations during symbolic execution, constraint solving, and finally, source code writing for test case generation.

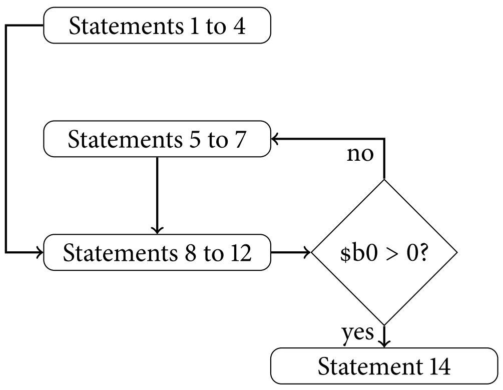
3.2.3 Constraint Solving

To keep the implementation in pure Java, initially CHOCO [41] was used for deciding the feasibility of paths and calculating solutions to error predicates. It provides a rich set of operators, corresponding closely to those available in the Java language, and it is able to reason over integral and real domains, mirroring the numeric types found in the JVM.

Unfortunately, we found that CHOCO did not scale particularly well. Instead of giving up after some time, CHOCO happily kept chugging along for days on end; we did not manage to get its own timeouts enforced during analysis. Therefore, we switched to a promising experimental constraint-solving framework, called Green [83]. It serves as a unified front-end to any constraint solver for which a translator is written (including CHOCO), and after bringing constraints into a canonical form, is able to cache results to prevent recomputation of known solutions.

```
0:  ldc2_w #16; //double 1.0E-15d
3:  dstore_2
4:  dload_0
5:  dstore 4
7:  goto 23
10: dload_0
11: dload 4
13: ddiv
14: dload 4
16: dadd
17: ldc2_w #18; //double 2.0d
20: ddiv
21: dstore 4
23: dload 4
25: dload_0
26: dload 4
28: ddiv
29: dsub
30: invokestatic #20;
    //Method java/lang/Math.abs:(D)D
33: dload_2
34: dload 4
36: dmul
37: dcmpl
38: ifgt 10
41: dload 4
43: dreturn
```

Figure 3.2: The bytecode for the method in Figure 3.1, produced by a standard Java compiler, and displayed by javap.



```
[ 1] d0 := @parameter0: double
[ 2] d1 = 1.0E-15
[ 3] d2 = d0
[ 4] goto [8]
[ 5] $d3 = d0 / d2
[ 6] $d4 = $d3 + d2
[ 7] d2 = $d4 / 2.0
[ 8] $d5 = d0 / d2
[ 9] $d6 = d2 - $d5
[10] $d7 = staticinvoke <java.lang.Math:
            double abs(double)>($d6)
[11] $d8 = d1 * d2
[12] $b0 = $d7 cmpl $d8
[13] if $b0 > 0 goto [5]
[14] return d2
```

Figure 3.3: The Jimple intermediate representation, produced by Soot, and a basic control-flow graph for the method bytecode in Figure 3.2.

3.2.4 Test Harness

JUnit [2] is a well-known and widely-used unit testing framework for Java. Version 4 is implemented by exploitation of the standard Java annotations framework. In practice, its basic functionality is not difficult to reproduce—Bloch [10] gives a rudimentary implementation of an annotation-based unit test framework—but we reasoned that a standard framework opened up additional possibilities for the tool, for example, as a part of a regression test suite.

Standard Java Development Kits (JDKs) include a Java interface to the JDK compiler, so that is used for the compilation of unit tests. Tests are run via the test runner component of the JUnit framework. To complete the test harness, we wrote an accounting subsystem that collates the test results.

As pointed out in the comments to Figure 1.3, there is a difference between running the analyses for bug-finding and for regression testing: For the former, triggering the exception found by symbolic execution during normal execution is viewed as a success; for the latter, not triggering that exception is a success. The accounting subsystem is, therefore, defined as an interface, with an implementation for bug-finding, but with the possibility of extension to other uses.

3.3 Artemis

Artemis was named after the homonymous Hellenic goddess of the hunt.[†] At the centre of the system lies a symbolic execution (SE) engine, which essentially defines and implements a virtual machine with SE semantics. In what follows, we have occasion to refer to both Artemis and the programs under analysis. To keep the concepts clear, we preface those that refer to the programs being analysed by "program", or sometimes, "symbolic". Unqualified concepts or those prefaced by "Artemis" refer to the tool itself.

3.3.1 The Symbolic Expression Hierarchy

A hierarchy of classes was written to represent the various components of symbolic expressions; refer to Figure 3.4. These expressions are used (i) to represent the values of variables during symbolic execution, (ii) to represent the constituent conjuncts in the path condition, and (iii) to formulate the constraints sent for solving. As such, they are defined as a tree structure to mirror the expression and value types defined by Soot's Jimple IR, which in turn mirrors the computations on the JVM stack machine.

The creation of this hierarchy, instead of using the Jimple structures as given, was motivated by the wish to keep the logic of the SE engine as uncluttered as possible. The engine proceeds over the Jimple statement types (see §3.3.3), whereas the hierarchy deals with expressions and values. Instead of writing extra logic to operate on the Jimple types directly or to compute the transformations listed below in the engine, it seemed simpler to localise logic. That is, the transformations and tests were added to those classes where they are most pertinent—to abstract classes for shared behaviour and to concrete classes for specific behaviour—and is exposed via a unified interface.

The list below provides an overview of the symbolic expression class hierarchy's salient features; details are provided in the following subsections.

1. The values being operated on by the SE engine can be either symbolic (that is, a constrained or unconstrained set of values) or concrete (that is, a single non-symbolic value, corresponding to a source code literal, or an expression consisting only of non-symbolic values). For simplification of expressions, any instance of a symbolic expression must at any given time know whether it is symbolic or concrete.

[†]She is also associated with childbirth and is sometimes called a protector of youth, both of which may be relevant for the programming discipline.

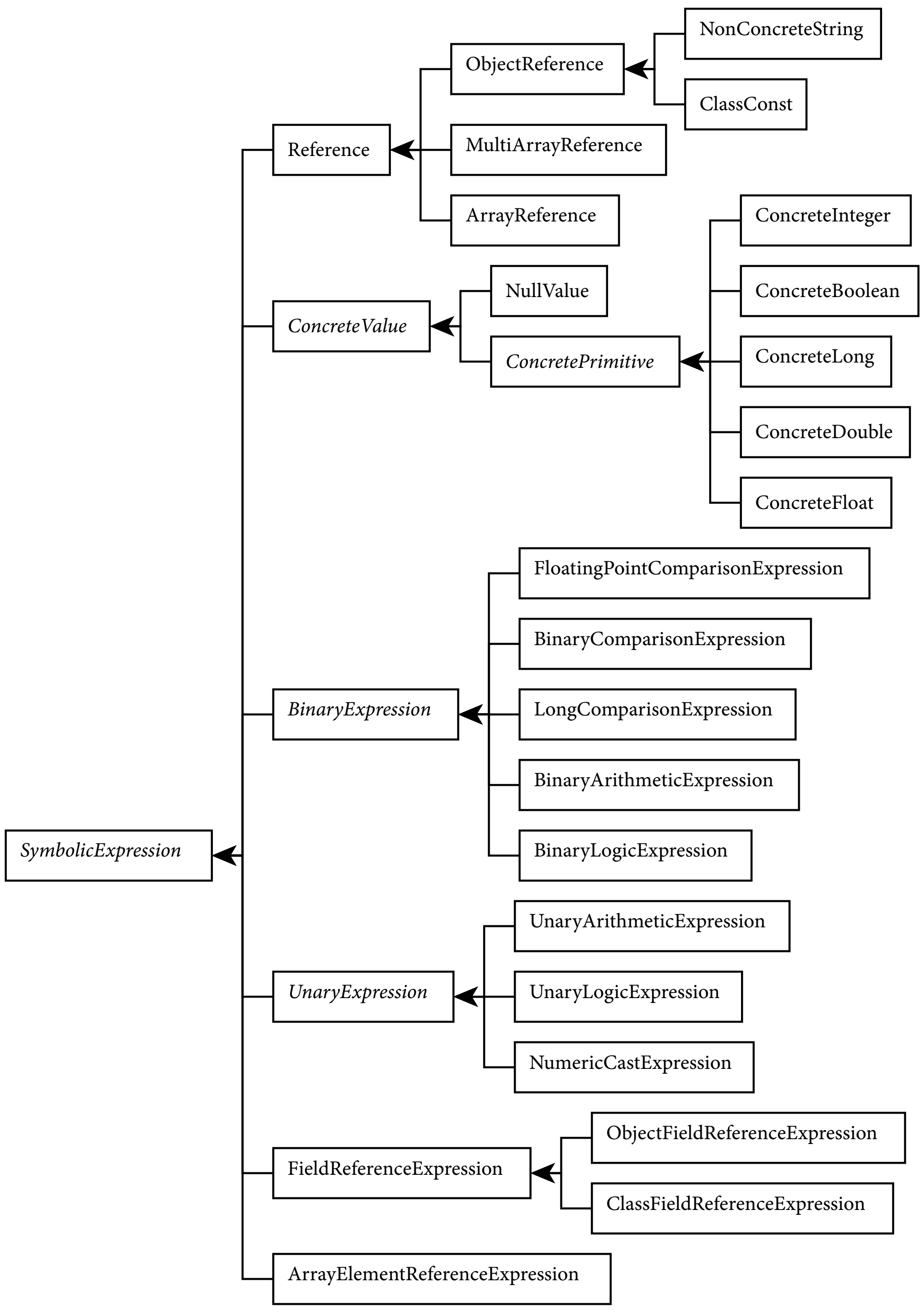


Figure 3.4: Inheritance diagram of the symbolic expression class hierarchy. The arrows give the EXTENDS relation. Class names in italics refer to abstract classes.

2. All values in the hierarchy are assignment-compatible. In particular, the representation of unconstrained (unknown) symbolic values can be assigned to the same structures for program variables as symbolic expressions.

3. To be able to handle branching over symbolic expressions, it must be possible for the symbolic execution engine to compute the logical negation of any given expression. The facility to do so was built into the hierarchy so that any expression can apply the relevant algebraic laws by recursion. Any operator knows its own negation operator locally. Operands, which may be expressions or atomic values, are handled by recursive calls, which stop when an atomic value is reached.

4. The error conditions we target are encoded into the expression types where they may occur during execution. As a result, the error handling could be mostly (but not entirely) separated from the engine.

Note that we use the term *hierarchy* to refer to the Artemis classes by which symbolic expression *trees* are represented. The salient features of this hierarchy are then that it represents a symbolic expression in a tree structure and that such a tree can be manipulated by calling a method that recurs on all vertexes in the tree.

3.3.1.1 The SymbolicExpression Class

The abstract class `SymbolicExpression` lies at the root of the symbolic expression hierarchy. It is used as the type for the expressions stored in state objects (see §3.3.2) and over which the engine operates (see §3.3.3). The class contains a number of static methods that receives objects of type `SymbolicExecution` as parameters: The hierarchy, therefore, mostly operates by overriding methods as opposed to overloading methods.

Jimple defines the following types of expressions: (i) binary arithmetic expressions (addition, division, multiplication, remainder, and subtraction); (ii) binary comparison expressions (equality, not equal, greater than, greater than or equal to, less than, and less than or equal to); (iii) binary logic expressions (bitwise AND, bitwise OR, bitwise exclusive-OR); (iv) array element reference expressions (retrieve a value from an array index); (v) expressions to cast operands to other types; (vi) expressions to throw and catch exceptions; (vii) comparison expressions for the `long` and floating-point types; (viii) constant values for the 32-bit and 64-bit signed integral and floating-point types, strings, as well as the `null` reference value; (ix) field references for class and object fields; (x) the `instanceof` operator; (xi) expressions to invoke methods; (xii) the `length` operator for the lengths of arrays; (xiii) expressions to load the value of a local variable and parameter; (xiv) a unary arithmetic expression for negation; (xv) expressions to create new objects, arrays, and multidimensional arrays; (xvi) expressions for signed and unsigned bitshifts; and (xvii) an expression to retrieve the `this` reference.

`SymbolicExpression` contains a nested class that translates these constructs into Artemis's symbolic expression hierarchy by application of the Visitor pattern.

The class provides static methods to test whether a type is (i) castable to another, (ii) floating-point, (iii) integral, (iv) numeric, (v) a reference, or (vi) what qualifies as a zero for that particular type. Where necessary, these methods forward their queries to the relevant classes in the hierarchy. Although the division is not exact in all cases, these methods tend to require knowledge of Soot types or of the program state, whereas the abstract methods of the following paragraph require local knowledge of how a particular symbolic expression is constructed. We do not shy away from using Java's `instanceof` operator, since we reason it is a fundamental feature of both the Java language and the JVM.[‡] Two last static methods trigger the simplification of base types (for objects and arrays) and numeric types that contain concrete values.

Since all methods inherit from `SymbolicExpression`, this class specifies abstract methods to (i) get the type of an expression, (ii) test whether an expression is concrete, and (iii) convert an expression to a Java `Object` for use during symbolic execution. In addition, the `toString()` method is overridden with an abstract method, and the `hashCode()` method is overridden to return the hash code of a symbolic expression's string representation. The string representations are defined to be unique identifiers in the subclasses, and they are also used during constraint solving and test case generation.

3.3.1.2 The ConcreteValue Classes

The hierarchy of classes rooted at `ConcreteValue` represent concrete (that is, non-symbolic) values. Since Soot translates variables into 32-bit and 64-bit integral and floating-point types, only the `int`, `long`, `float`, and `double` Java numeric types are represented. The class `NullValue` represents the `null` reference, and is implemented as a singleton, eagerly initialised. In addition, the class `ConcreteBoolean` provides support for `boolean` types. All of the `ConcretePrimitive` subclasses support limited caching for commonly used values.

3.3.1.3 The BinaryExpression Classes

The classes that extend the class `BinaryExpression` all represent infix operators that take left and right operands. For operands that simplify to concrete values, the `toObject()` methods of these classes compute concrete results.

1. The class `BinaryArithmeticExpression` is for expressions over the arithmetic operators for addition, subtraction, multiplication, division, and computing the remainder (alternatively, modulo); both the integral and the floating-point types are handled. The class checks for integer division by zero.

[‡]Those who object to its use in the basis of efficiency and speed should do well to remember that the same objections can also be lodged against the JVM's resolution of dynamic references via strings in the constant pool.

2. The class `BinaryComparisonExpression` handles expressions for the equal to, not equal to, less than, less than or equal to, greater than, and greater than or equal to relational operators. It can handle both integral and floating-point types, although the latter only appears during symbolic execution, and never directly from the program under analysis; the reason for this is given under item 4 below. This class also contains logic to compute the logical negationegation of any given expression; which is used when marking errors.

3. The class `BinaryLogicExpression` handles expressions for the signed left and right bitshifts, the unsigned right bitshift, the bitwise AND, OR, and exclusive-OR operators, as well as logical (boolean) AND and OR operators. The latter two are not defined in the JVM—instead boolean operations happen over integer values and the bitwise operators—but they do arise naturally during symbolic execution, and make translation to a format for constraint solving easier.

4. Only the integral types have test-and-branch instructions defined in the JVM. The `long` and floating-point types use the idiom where −1, 0, or 1 is pushed onto the stack depending on whether the left operand is, respectively, less than, equal to, or greater than the right value. On the JVM, branching can then occur by (integer) comparison to zero, which in Artemis is handled by the `BinaryComparisonClass`.

5. The `LongComparisonExpression` class represents the comparison idiom described in the previous item for `long` values.

3.3.1.4 The UnaryExpression Classes

Three classes inherit from the class `UnaryExpression`, and they represent operators with a single operand. The class `UnaryArithmeticExpression` handles the unary minus (arithmetic negation), and `NumericCastExpression` handles widening and narrowing conversions between numeric values. The class `UnaryLogicExpression` provides a logical `not` operator which, although not present in the JVM, is used when inverting logical expressions during symbolic execution. Finally, the class `ArrayLengthExpression` handles the `length` operator on array references, and therefore checks for the reference being null.

3.3.1.5 The Reference Classes

The class hierarchy rooted at the `Reference` class represents JVM reference types for object, array, and multi-array references. It may be interesting to note that, under symbolic execution, references can also be viewed as either symbolic or concrete. When an object reference is the result of a `new` bytecode instruction, it is viewed as being concrete and non-null; we reason here that failure to produce a newly allocated object on execution of these instructions is not an error of execution semantics, but rather an error with the JVM itself, for example, running out of memory. References that arise from other contexts, for example, from parameter passing to top-level methods, are treated as symbolic.

Array references are treated slightly differently since they require dimensions for creation with the `newarray`, `anewarray` (for arrays of references), or `multinewarray` instructions. Only if an array reference is created by one of these "new" statements and the dimensions are concrete is the array reference treated as concrete. All other cases result in symbolic array references. The `ArrayReference` class checks for negative array size errors.

Note that concrete references can never be null; these are represented by the `NullValue` class. Also, null references only result in errors—for symbolic execution, at least—when the base object of a field is dereferenced; they are therefore checked in the classes of §3.3.1.6. Finally, all references have unique string identifiers that are used during constraint solving and test case generation.

3.3.1.6 The FieldReferenceExpression Classes

The two classes that inherit from `FieldReferenceExpression`, namely, `ObjectFieldReferenceExpression` and `ClassReferenceExpression`, represent instance and static field references, respectively. Since classes are loaded as soon as they are referenced during normal execution, null-pointer dereferences can only occur on instance fields, and `ObjectFieldReferenceExpression` checks for safety. All reference expressions are cached in maps from Soot reference expressions to symbolic reference expressions for use during program state updates.

3.3.1.7 The ArrayElementReferenceExpression Class

The class `ArrayElementReferenceExpression` represents the reference of an array element by index. For this, two errors are checked: the array reference being null and the index being out of range.

3.3.1.8 The UnkownValue Class

The `UnknownValue` class represents atomic unknown symbolic values, that is, to which no constraint has been applied. In essence, they model the input to a program, whether or not this input results from values passed as parameters or by reading values from a field that would have been set up by other means, for example, setter methods not currently under analysis. They are also the values for which we expect solutions by the constraint solver, and which we use for parameters and assignments during test case generation.

The class keeps a (static) map of identifiers to unknown values so that particular unknown values can be retrieved for constraint solving and code generation. To ease the latter, each unknown value may also keep a reference to its storage, that is, the field whence it was referenced first. Note that each time an unknown, uninitialised value is encountered *during symbolic execution*, a new `UnknownValue` instance is created.

3.3.2 Program State

Program state is kept via instances of the `SymbolicState` class, and caters for both symbolic (program) state and Artemis state. As described in §3.3.3, a state object is associated with each path explored. In particular, once symbolic execution indicates the possibility of an error, the constraints leading to the error are added to the state, and the state object in its entirety is handed off to the constraint solving subsystem. Each state object is therefore a complete record of the SE path, and also a snapshot of the program state when a possible error is encountered.

All the state variables are declared as `private`, and can only be accessed via methods intended to limit side effects. However, it is the state objects that contain the methods which trigger the search for solutions, and if they are found, add the solutions to the test case source code writer for emission.

To keep program state for field values and array entries, maps are provided from reference expression types to values of type `SymbolicExpression`; initial as well as current entries are stored. Since the co-domain of these maps is symbolic, lazy initialisation, from an SE viewpoint, is employed: A field or array entry only has a (symbolic) value once it is assigned to or used during symbolic execution. Local variables are mapped from the Soot `Local` type—for that is how they appear in the statements executed by the SE engine—and they are also lazily initialised. Bases—array and object references—are simplified when they are updated. Sequences of initial and current parameter values are also stored.

The symbolic state also stores the basic parameters of symbolic execution as given by King [46]: The path condition (as a list of symbolic expressions) and the program counter (pointing to the current statement of the method being executed). The branch counters, to limit execution tree depth, is kept as a map of branch statements to integers. To enable test case generation for interprocedural analysis, where the test parameters must be generated for top-level methods, both the (top-level) method at which analysis starts and the one "lower down" where a possible error is found, are preserved in the symbolic state.

The last group of variables concerns Artemis's execution parameters, bookkeeping, and context information of errors. Since a state object is associated with each execution, and since each possible error context found is treated as a separate path, only one error is allowed per object. The execution parameters specify (i) whether, on location of an error, the constraint solving subsystem is called, (ii) whether the predicate being added to the path condition is checked for obvious contradictions (that is, if both the symbolic expression q and $\neg q$ is present), and (iii) whether paths are pruned (that is, the constraint solver is called to decide feasibility as paths are added). These parameters are set when the analysis is first started.

3.3.3 Symbolic Execution Engine

Artemis's SE engine is capable of symbolically executing a method given in the Jimple IR. Each method under analysis is analysed in its own instance of the `SymbolicExecutor` class. In this respect, the class implements the SE semantics for a single stack frame in the JVM. An instance contains a number of ancillary structures to handle (i) a method's Jimple representation, (ii) traps (jumps to code for exception handling), (iii) branching over symbolic branch guards, and (iv) multiple return states for interprocedural analysis.

Soot defines a data structure, called a **chain**, to carry the Jimple IR of methods. It is the amalgamation of a map and a list, defining methods to access a particular method statement (called a *unit* by Soot) by identifier or by position through a successor/predecessor relation. Traversal of a method chain using our SE semantics is tantamount to traversing a control-flow graph of the method; see Figure 3.3 for a simple control-flow graph of Jimple code.

We apply the Visitor pattern to interpret the following Jimple statement types: (i) assignments, (ii) break points, (iii) entering and exiting monitors, (iv) unconditional jumps, (v) identity statements (associating parameters with the local variable array), (vi) conditional branching, (vii) method invocation, (viii) lookup and table switches for conditional branching on more than one branch guard, (ix) no operation, (x) returning from methods and exception handlers, and (xi) explicitly throwing an exception. Statements such as those for assignments follow default flow in that the next statement to be executed is given by the ordering of Jimple units within the method chain. Other statements, such as those for jumps and branching, specify their own (multiple) successor statements.

The execution of some statements is not particularly interesting. For example, an assignment statement requires no more than constructing symbolic expressions from our hierarchy—by calling the single static `getSymbolicExpression` method in the class `SymbolicExpres sion`, which translates Jimple to the hierarchy—and then associating the symbolic representation of the right-hand side of the assignment with the variable on the left. Handling conditional branching, method calls, and unhandled exception propagation is more involved.

3.3.3.1 Conditional Branching during Symbolic Execution

Branching for looping statements is reduced, during the original code generation, to combinations of **if** and jump statements.[§] The detail in this section, therefore, applies equally to normal and loop branching, with the switch statements natural extensions to the mechanism where guards are matched with respect to table entries.

[§]It is interesting to note in passing that boolean short-circuiting of a branch guard is accomplished in the same way by the Java compiler. The JVM itself has no notion of short-circuiting.

A Jimple **if** statement specifies a target for a **true** evaluation of the guard, and the **false** case is handled by fall-through, that is, continuing with the statement's successor as defined by the method chain. If the branch guard is concrete, symbolic execution simply follows the appropriate path.

However, if the branch guard is symbolic, both paths must be explored. In the latter case, the current state object is cloned so that both branches start with the same program state. For a branch guard q, the symbolic expression for q is conjoined to the **true** path state object, and that for $\neg q$ to the **false** path state object. Then both paths are explored, one after the other, with recourse to a stack that temporarily stores the state objects of unexecuted branches. This choice over symbolic guards is only allowed if the branch counter (in the state object) associated with that particular branch point statement is greater than zero. Since each branch point results in the cloning of a state object—in essence, "unwrapping" the method's control-flow graph into an execution tree as computation progresses (see Figure 1.2)—and for each branch point, only the counter associated with that branch point is decremented, nested loops are handled correctly, that is, for a nested loop with branch bound n, the inner statements are executed n^2 times.[¶]

The branch bound is specified when the symbolic executor for a method is first instantiated, and is therefore propagated to invoked methods. To prevent infinite loops over concrete guards, bounds may also be applied to these loops. Once the state objects for branches diverge, they are never merged with respect to "completing" a branch, because they now characterise different paths through the method. Indeed, it is this separation that allows proper loop unrolling in the first place.

3.3.3.2 Method Invocation for Intraprocedural Analysis

An interesting case for symbolic execution is handling method invocation. The depth of method calls is controlled by a variable, which is decremented when a new `SymbolicExecutor` is instantiated during a method call: Every executor is called with a depth parameter of one less than that of its parent. Once zero is about to be reached, further method calls on a particular level are prevented, and the result when a call is prevented is a new instance of the `UnknownValue` class. Calls are also prevented and an unknown value "returned" whenever no active method body is available (for example, a method from a library that was not included for analysis) or when the method is native.

For each method, a number of paths may exist. This implies that a set of state objects may be returned by any particular instance of `SymbolicExecutor`. Such return states are merged one by one into the current branch state object of the calling executor, in particular, to ensure that parameter expressions are reset to appropriate values. If a return state case includes a live exception, that is, one not handled in the method that was called, the caller's exception

[¶]This is in contrast with Tomb et al. [79], where counters appear to be associated with statements.

handlers are examined. If a matching handler is found, that handler is called; otherwise, the exception is left live for propagation up the call stack.

The `SymbolicExecutor` class is the only class outside the symbolic expression hierarchy that checks for an error. For instance methods, the containing object may be null, so checks are included for both concrete and symbolic cases.

3.3.4 Constraint Solving

Artemis treats the decision procedure and constraint solver as a single entity. Since we are interested in writing source code for a unit test that forces an error at a particular statement, a decision result of being satisfiable without an assignment of values to model the case is useless. An implementation of our `SolutionFinder` interface acts as translator to and from a given constraint solver. We have supplied two implementations, one directly to the Choco solver, and the other to the Green framework. Translation to a particular format for constraint solving happens by applying the Visitor pattern to the symbolic expression hierarchy. To handle the tree structure, a stack is used as intermediary storage while the constraints are built up from the symbolic expressions.

The constraints that must be solved consist of the path condition plus a predicate that characterises a particular error. This predicate may be in one of two forms: If the path condition is satisfiable and leads to a specific statement point in the execution, either an error will be definitely triggered (for example, we know that the base object of a field reference is definitely null), or an error may be triggered if an additional constraint is satisfied (for example, we don't know whether the base object of a field reference is null). For the latter case, when the statement in possible error is reached during symbolic execution, the current state object is once again cloned: Constraints for a "good" path added to the original, and constraints for a "bad" path added to the clone, which is then handed to the constraint solving subsystem. The reason is that if the constraint characterising the error cannot be satisfied, symbolic execution must be allowed to continue exploration of that particular path. For errors for which we know that satisfiability is sufficient to trigger an exception, only the path condition is sent for solving.

The following symbolic contexts generate errors; refer to §4.1 for examples of how Artemis finds such errors:

1. Dereferencing of a null reference for an access to a field, calling an instance method, or querying the length of an array: For the reference b, the good path is characterised by

   $$pc \wedge b \neq \texttt{null}, \tag{3.3.1}$$

   and the bad path by

   $$pc \wedge b = \texttt{null} \tag{3.3.2}$$

   for a `NullPointerException`.

2. Accessing an array element by index: For the array reference a and index i, the good path is characterised by

   $$pc \wedge i \geqslant 0 \wedge i < a.\texttt{length}, \tag{3.3.3}$$

   and the bad path by

   $$pc \wedge (i < 0 \vee i \geqslant a.\texttt{length}), \tag{3.3.4}$$

   for an `ArrayIndexOutOfBoundsException`. The array length property is handled by the `ArrayReference`. It can be concrete or symbolic, and is handled by the emission of a concrete or unknown value when constraints are translated to the solver's representation. For example, for a symbolic length and index, we need solutions for both to be able to trigger the error during normal execution.

3. Creating a new array reference with a negative length: For the supplied array length ℓ, the good path is characterised by

   $$pc \wedge \ell \geqslant 0, \tag{3.3.5}$$

   and the bad path by

   $$pc \wedge \ell < 0 \tag{3.3.6}$$

   for a `NegativeArraySizeException`.

4. Integer division by zero: For the integral expressions x/y and $x \bmod y$, the good path is characterised by

   $$pc \wedge y \neq 0, \tag{3.3.7}$$

   and the bad path by

   $$pc \wedge y = 0 \tag{3.3.8}$$

   for an `ArithmeticException`.

If a solution to given constraints was found by the constraint solver, the details are handed off to the test case generation subsystem. In some cases, certain conjuncts in a constraint cannot be handled by the underlying theory of the selected constraint solver. In such cases, those particular conjuncts are removed from the path condition, and what is left of the path condition is solved again in the hope that the remaining conjuncts are sufficient to trigger the exception.

3.3.5 Test Case Generation

For the test cases, we maintain a one-to-one mapping between a class file under symbolic execution and a JUnit test class. When a solution passes to the test case generator, it is first stored in a temporary structure associated with the class under analysis. When the entire analysis is finished, the JUnit source is written out, compiled, and run. Note that we treat only one error per test method in the JUnit class. If more than one (possibly different) errors exist on different paths through a method, different test methods are generated. As solved error constraints are added, sequence numbers are attached so that the method names can be both descriptive (using the name of the method under analysis) and unique.

For intraprocedural analysis, test methods are generated for the top-level methods that lead, following method calls, to the error context. It is to this effect that each state object contains a reference to the top-level method in its call sequence. Also, each test solution is checked whether it involves assignments to fields declared as `final`. Such cases are recorded through the accounting mechanism, but no source will be generated as compilation will fail.

When symbolic execution has finished, Artemis collects information on the constructors of those classes in which methods where found to contain errors. Then it iterates over these methods, for each creating a new writer that iterates over the collected error contexts for that method, and dumps properly formatted JUnit source code via the StringTemplate library.

Public methods with primitive parameters are simplest to handle. Via the information in a state object, the solution for each unknown value associated with a parameter is used when writing the method call. Where no such association exists for a parameter, a value is randomly generated from the applicable type domain.

Instance methods and methods that take reference parameters first require that suitable objects must be instantiated. We take the approach that constructors with the fewest parameters are used, which in many cases, implies use of a default parameterless constructor. Fields are populated by recurring over the parameter types of the method being tested.

To mitigate problems with member visibility, the JUnit tests are written to a separate source tree that mirrors the package structure of the program under analysis. Therefore, only fields declared as `private` remain to be subjected to a visibility change via Java's reflection mechanism. Should a class, however, employ appropriate security features, and the test cases are run on a standards-compliant JVM, testing can still fail. In the end, we decided not to "publicise" `private` fields in any way, and the code generated to handle private methods is primitive to say the least.

The code to trigger an error is wrapped in a **try** statement with two `catch` clauses: The first handles the expected exception, and the second records any other exception that has occurred. Details of the error context is written to the source code file. In particular, the path condition and solutions not used to trigger errors are inserted as comments for later analysis by hand.

3.3.6 Test Harness

After the source code for tests has been written out, Artemis calls the Java compiler included in the JDK, and then starts the tests via the `JUnitCore`. The JUnit test methods report results via Artemis's `Accounting` class. The following cases are distinguished for accounting:

1. the expected exception was successfully triggered during execution of the test case;
2. a path and error constraint could not be solved;
3. the path was pruned after finding (in the SE engine) that the negation of a conjunct about to be added is already present in the path condition;
4. a path was pruned after a check by the constraint solver when a new conjunct is about to be added;
5. the test case found a possible error in a static class initialiser, which we cannot reach with a unit test;
6. the constraint solver could decide the path condition, but did not return the variable assignments we need during test case generation;
7. the maximum path condition length was reached, if such a maximum is enforced;
8. a solution run in a test case did not trigger the expected exception;
9. during test case generation, assignment to a field would have violated the `final` specifier;
10. test case generation failed for any other reason; and
11. a different exception than the one indicated by analysis was triggered when the test case was run.

A method to display the results is provided. Its output includes (i) run-time measurements, (ii) the number of state objects created (an indication of the number of paths explored), (iii) the number of state objects that signified errors, and (iv) the length of the longest path condition handled.

3.4 Observations and Experiences

3.4.1 The Soot Framework

The use of the Soot framework was initially motivated by its convenient Jimple IR. However, it suffers from major deficiencies from our point of view: (i) It lacks proper source code documentation, (ii) it was written before generics were introduced into the Java language, and (iii) none of the classes are serialisable. The first deficiency made it difficult to use only what we needed, and the second meant using a lot of annotations suppressing warnings for raw types, cluttering our own code. Also, although we presume the original design to have been according to the state of the art a decade ago, it appears that years of academic experimentation and tinkering to add new analyses have left the code in some need of clean-up. However, we do realise that we misused Soot to a great extent, in particular when trying to stay separate from Soot's own workflow mechanism, which made it particularly tricky, amongst other things, to make sure that the Jimple representations have been built when required by Artemis.

3.4.2 Constraint Solving

Although not part of the investigative domain of this dissertation, constraint solving presented some headaches. Initially, CHOCO performed well for small examples and limited path conditions. It also supplied a rich set of operators and expressions corresponding to those found in the JVM, and we were eager to use a pure Java solution. However, on larger examples CHOCO simply got lost in computation—which led to a colleague calling it "choke-oh"—on what turned out to be infeasible paths. Attempts to limit its running time, both by internal (its own mechanisms) and external (JVM-based) means, were not successful.

The switch to the Green framework speeded analysis up tremendously, in particular, when failing fast on infeasible paths. However, it is still very much experimental, so that we stayed with underlying solvers which do not provide all the required operations found often in the code we analysed. Here, the absence of the modulo operator was a particular annoyance.

Green reports constraints that cannot be handled only after it has attempted to solve them, and then only by an exception from the underlying solver. This lead to a stopgap measure of testing constraints one by one after a path condition has resulted in an exception, and removing troublesome constraints from the path condition, the idea being that we can still try to generate test cases in the hope that sufficient conditions to identify a path are present in what is left.

Finally, it turned out that Green does not yet handle disjunction correctly. A workaround for the expression $pc \wedge (p \vee q)$ was, therefore, necessary: We solve two separate constraint expressions $pc \wedge p$ and $pc \wedge q$, and generate test cases for both if satisfiable.

3.4.3 Test Case Generation

The test case generation subsystem resulted in many a light night. It went through a major rewrite halfway into the project, when we switched from writing Java statements directly from a Java class to dumping the source via a template system. This resulted both in cleaner output and being able to focus on the logic of setting up the appropriate state, separately from formatting the test case.

Still, the test case generation subsystem remains a tricky business; a couple of examples follow:

1. Many of our test cases for `NullPointerExceptions` required `null` references to be used as method arguments. However, when a class under test contained overloaded methods with the same reference signature—for example a method `foo` is overloaded on both a `String` and an `Object` parameter—the Java compiler called from within Artemis could not distinguish between them. When some of our larger test runs failed to trigger expected exceptions, we initially assumed the error state is simply not reachable via a test case. Until, that is, we eventually realised method overloading is in play. Our basic solution was to cast each `null` reference to the Java type listed for the method that we were targeting.

2. In similar vein, when a parameter of a targeted method has an abstract type that must be non-null for a test case, the problem arose of finding a suitable implementation or extension to construct. For frequently encountered standard classes, we therefore keep a list of suitable parameterless constructors. However, for abstract types written as part of the code under analysis, the problem remains. This is an obvious flaw in our implementation, and something we need to address in the future. We built a working class traversal mechanism to compute the inheritance graph of the classes under analysis—using reflection did not help, since one can only walk towards superclasses, and we needed extensions or implementations of the abstract type—but its use only complicated the test case generation when we were pressed for time.

3. We currently select constructors for the smallest number of parameters; when there is an equal choice between a reference type and a primitive type, we prefer the primitive type, as it does not require the reference type to be constructed in turn. This situation is far from ideal, and more complete solutions would be to try and link the symbolic execution of available constructors to the symbolic execution leading to an error, or at least, to try out different constructors in difference test source files.

4. Initially, we collected all test methods for a targeted class into one JUnit class. However, once we made our peace, for now, with not being able to generate all test cases successfully, we expected compilation of some test methods to fail. Therefore, we added the functionality both to generate test classes containing only one method, and the ability to serialise and store the information necessary to create a test case. Since the Soot classes we used for symbolic state information are not serialisable, this results in having to rewrite a number of classes to be able to store the information in string format.

3.4.4 Overengineering and Java in General

In hindsight, we perhaps were too eager to follow the advice of Bloch [10]. This kind of orderliness comes at a price: It is relatively easy to add orthogonal behaviour to a well-designed interface, but is another matter entirely when the assumptions on which the initial design was based are found to be incomplete, or even incorrect. Such, we presume, is the nature of research code.

For example, following Soot, we distinguished from the start between object and array references. In particular, we assumed that no object can be indexed, and this is borne out by our “tyre kicking” examples, and even some larger tests. We came upon an example where an array reference is returned as an object reference, by using Java’s cloning mechanism. It necessitated a transfer for indexing methods up the symbolic expression hierarchy, something we were initially loathe to do, since to check our symbolic execution engine, we enforced assertions on what actions are allowed on which particular kind of symbolic value.

Had we been surer of the correctness of the implementation (that is, understood the theory better during the initial design), we would have made some better design choices. In our experience, Java is not particularly forgiving to a piecemeal design approach. A lack of implicit default behaviours meant a lot of scaffolding was necessary before any real work could be attempted. This will weigh heavily on future extensions to Artemis.

Chapter four

Results

In this chapter, we present our results in three stages:

1. We analysed a number of Java programs to test the basic functionality of our tool Artemis. In particular, for this "tyre kicking" exercise, we seeded each Java source code file with errors that we expected Artemis to be able to find and for which it should be able to generate test cases successfully.

2. As a sanity check, we then analysed a set of programs from the JCrasher website[*]. Written by student programmers, these programs were originally introduced by Csallner and Smaragdakis for the evaluation of JCrasher [23], and subsequently re-evaluated by Tomb et al. [79]. An initial version of our results [9] was successfully presented at the SAICSIT conference held on 1–3 October 2012, in Centurion, South Africa.

3. Finally, we analysed most of a large program, the Java PathFinder [1], excluding from analysis only those parts that rely on non-trivial I/O (for example, from files, for which we cannot generate tests) and suffer from some other issues during analysis (which we describe in §4.3).

Experimental setup For our evaluation, we employed two different machines:[†]

1. We designate as *Machine 1* a system with two dual-core, hyper-threading enabled 2.27 GHz Intel Xeon processors, running the Linux 2.6.18-308.4.1.el5 kernel on CentOS, where the JVM performing the experiments has access to 4 GB of memory.

2. We designate as *Machine 2* a system with one quad-core, hyper-threading enabled 3.07 GHz Intel Core i7 processor, running the Linux 3.2.0-54-generic kernel on Ubuntu 12.04, where the JVM performing the experiments has access to 2 GB of memory.

Both machines run Oracle Java version 7, although the source levels of both Artemis and the programs analysed were set to version 6.

[*] `http://ranger.uta.edu/~csallner/jcrasher/`
[†] These specifications were calculated as described by Smith [75].

```java
public class ZeroDivisor {

  public static int div(int x, int y) {
    return x / y;
  }

  public static int mod(int x, int y) {
    return x % y;
  }

  public static int test(int x, int y, int z) {
    if (3*x + 5*y < 100 && y > 3 && y < 20)
      return x / (y + z);
    else
      return z;
  }

}
```

Figure 4.1: A Java class with methods containing possible zero divisor errors.

Code metrics Where given in the sequel, code metrics were computed using the Unified Code Count (UCC) tool [3] by the Center for Systems and Software Engineering of the University of Southern California's Viterbi School of Engineering. In particular, we often use

1. *logical lines of code* (LLOC), which disregards blank and comment lines, and which uses Java-specific definitions of what constitutes an executable line of code; and

2. *cyclomatic complexity* [56], a measurement of the number of independent linear paths through the code, to gauge the complexity of a program's branching structure.

4.1 Small Examples to Illustrate Finding Errors

This section presents working examples illustrating the application of Eqs. (3.3.2), (3.3.4), (3.3.6), and (3.3.8) to locating possible errors with symbolic execution. We also give an example of an exception explicitly thrown by the code under analysis. In each case, we give the Java source code of the method(s) being considered, as well as the test case source code generated by Artemis.

4.1.1 Division by Zero

Figure 4.1 contains a class with three static methods: The first two methods wrap statements to compute the integer quotient and integer remainder, respectively, of two integer operands,

```
import za.ac.sun.cs.artemis.exec.Accounting;
import org.junit.Test;
import static org.junit.Assert.*;

public class ZeroDivisorTest {
  @Test public final void testDiv() {
    // pc: (uv_int1 == 0)
    try {
      ZeroDivisor.div(/* uv_int0 */ 2098754319, /* uv_int1 */ 0);
      Accounting.failure("SS@6855a338");
      fail("Expected␣ArithmeticException␣not␣thrown␣on␣line␣4.");
    } catch (ArithmeticException e) {
      Accounting.success("SS@6855a338");
      assertTrue(true);
    } catch (Exception e) {
      Accounting.unexpected("SS@6855a338", e);
      fail("On␣" + e.getClass() + ":␣" + e.getMessage());
    }
  }
  @Test public final void testMod() {
    // pc: (uv_int3 == 0)
    try {
      ZeroDivisor.mod(/* uv_int2 */ 1036425361, /* uv_int3 */ 0);
      Accounting.failure("SS@2f87c55c");
      fail("Expected␣ArithmeticException␣not␣thrown␣on␣line␣8.");
    } catch (ArithmeticException e) {
      Accounting.success("SS@2f87c55c");
      assertTrue(true);
    } catch (Exception e) {
      Accounting.unexpected("SS@2f87c55c", e);
      fail("On␣" + e.getClass() + ":␣" + e.getMessage());
    }
  }
  @Test public final void testTest() {
    // pc: (((3 * uv_int4) + (5 * uv_int5)) < 100)&&(uv_int5 > 3)
    // &&(uv_int5 < 20)&&((uv_int5 + uv_int6) == 0)
    try {
      ZeroDivisor.test(/* uv_int4 */ -2147483648,
                       /* uv_int5 */ 4, /* uv_int6 */ -4);
      Accounting.failure("SS@6632060c");
      fail("Expected␣ArithmeticException␣not␣thrown␣on␣line␣13.");
    } catch (ArithmeticException e) {
      Accounting.success("SS@6632060c");
      assertTrue(true);
    } catch (Exception e) {
      Accounting.unexpected("SS@6632060c", e);
      fail("On␣" + e.getClass() + ":␣" + e.getMessage());
    }
  }
}
```

Figure 4.2: The JUnit test case source code generated by Artemis for show the presence of errors in the `ZeroDivisor` class of Figure 4.1, slightly reformatted to fit the page.

```java
public class ArrayAccess {
  public static int get(int[] a, int i) {
    return a[i];
  }
}
```

Figure 4.3: A Java class with methods containing possible null-pointer and array index-out-of-bounds errors.

and the third a slightly less trivial example that includes an **if** statement and an expression as a divisor. Artemis correctly determines that errors of the class `ArithmeticException` are possible in lines 4, lines 8, and 13. To confirm that these errors can indeed be triggered by a execution over concrete input values, Artemis generated the JUnit-based test class given in Figure 4.2 following symbolic analysis.

We examine the test case for the method `div` in some detail; the test case for `mod` is similar. The method `div` contains only one execution path, so that the path condition is identically true. The path condition as determined by Artemis, given as a comment in line 7, therefore consists of a single expression of the error constraint given by Eq. (3.3.8). The unknown values `uv_int0` and `uv_int1`, respectively, were associated with the two parameters of `div` during symbolic execution. The constraint solver provided a solution of zero for `uv_int1`. Using this solution, and choosing a random `int` value for `uv_int0`, allowed Artemis to construct the method call in line 9. It is written inside a **try** clause in the expectation that an exception will be produced by the call, which we can then catch in line 12. Reaching lines 10 and 11 during test case execution is viewed as the error being spurious for the particular path condition, that is, the exception expected in line 9 could not be produced during normal execution. Accounting is with respect to a unique identifier for each symbolic state, and we include an extra **catch** clause in line 15 to account for any unexpected exceptions.

The method `test` in Figure 4.1, contains more than one path, one of which leads to a possible exception in line 13. Therefore, Artemis's path condition in lines 35 and 36 of Figure 4.2 contains both a conjunct for the path leading to the error statement and an expression for Eq. (3.3.7). In this case, all unknown values associated with method parameters required solving, and the solutions were used in the method call in lines 38 and 39 of the test case.

For all three methods, execution of the test case methods results in the expected error, and therefore Artemis flags these errors as real.

4.1.2 Null References and Array Access

Figure 4.3 contains a class with two static methods, wrapping access to an array via values passed as method parameters. For the method `get`, Artemis determines two errors to be possible: a null reference on the array [Eq. (3.3.2)], and an index out of bounds [Eq. (3.3.4)].

```
// pc: (uv_int0 > 0)&&(arr0 == null)
ArrayAccess.get(/* arr0 */ (int[]) null,
                /* uv_int1 */ -338367592);

// pc: (uv_int0 > 0)&&(arr0 != null)
// &&((uv_int1 < 0) || (uv_int1 >= uv_int0))
int[] arr0 = new int[1]; // array ref: arr0; length: uv_int0
ArrayAccess.get(/* arr0 */ arr0, /* uv_int1 */ 1);
```

Figure 4.4: The **try** clauses of the test cases generated for the get method in Figure 4.3.

```
public class ArrayCreation {
  public static int[] newArray(int n) {
    return new int[n];
  }
}
```

Figure 4.5: A Java class with a method containing a possible negative array length error.

```
// pc: (uv_int0 < 0)
ArrayCreation.newArray(/* uv_int0 */ -2147483648);
```

Figure 4.6: The **try** clause for the test generated for the newArray method in Figure 4.5.

Therefore Artemis generates two test cases, the **try** clauses of which (for brevity) is given in Figure 4.4, identified by the path conditions in comments.

First, Artemis determines that a `NullPointerException` is possible for the array reference passed in as parameter, so it generates a test on a `null` reference value, and passes a random value for the index, as given in the first two lines of Figure 4.4. Note the cast for `null` references to ensure that overloaded methods with reference parameters can be dynamically resolved by the Java run-time environment. Second, Artemis determines that, when the array reference is known to be non-null, a `ArrayIndexOutOfBoundsException` is possible for indexed access. For this case, the path condition contains an expression constraining the array reference to be non-null, but also for an index out of bounds with respect to the array length. Note that, before the method call, Artemis first includes an array creation statement. When normally executed, both exceptions are successfully triggered, and therefore, reported as real.

```
public class Primes {

  public static void primes(int n) {
    if (n < 2)
      throw new IllegalArgumentException("n < 2");
    for (int i = 2; i*i <= n; i++)
      while (n % i == 0) {
        System.out.print(i + " ");
        n = n / i;
      }
    if (n > 1) System.out.println(n);
    else       System.out.println();
  }

}
```

Figure 4.7: A Java class that signals an illegal argument with an exception.

```
@Test public final void testPrimes() {
  // pc: (uv_int0 < 2)
  try {
    Primes.primes(/* uv_int0 */ 0);
    Accounting.failure("SS@2e01787");
    fail("Expected␣IllegalArgumentException␣" +
          "not␣thrown␣on␣line␣5.");
  } catch (IllegalArgumentException e) {
    Accounting.success("SS@2e01787");
    assertTrue(true);
  } catch (Exception e) {
    Accounting.unexpected("SS@2e01787", e);
    fail("On␣" + e.getClass() + ":␣" + e.getMessage());
  }
}
```

Figure 4.8: The JUnit test method, slightly reformatted to fit the page, that was generated for the method `primes` in Figure 4.7.

4.1.3 Negative Size for a New Array

Figure 4.5 contains a Java class with a method that creates a new array. Array creation on a negative length results in a `NegativeArraySizeException`. The **try** clause for a test case, which triggers this exception during normal execution, is given in Figure 4.6.

Table 4.1: Code metrics for the `P1` programs analysed in §4.2.

Student	Lines	LLOC	Methods	CC[a]		Known Errors	
				Max	Avg	By [23]	By [79]
s1	503	109	17	4	2.67	5	6
s1139	462	115	16	5	2.86	3	8
s2120	383	93	17	5	2.60	5	5
s3426	439	106	19	4	2.59	8	11
s8007	376	101	16	6	2.92	1	2

[a]Cyclomatic Complexity

4.1.4 An Exception Thrown Explicitly

Figure 4.7 contains a class, adapted from Sedgewick and Wayne [71], that contains a single method to compute and print out the prime factors of a specified integer n. Since the implementation produces a prime factorisation only if $n \geqslant 2$, the method is checked by a guard that tests whether this is the case; if this test fails, an `IllegalArgumentException` is thrown. Artemis determines that an assignment of zero to n will direct the path of execution to the `throw` statement in line 5. The JUnit test method that successfully triggers the `IllegalArgumentException` is given in Figure 4.8.

It is both interesting and instructive to note that whether or not this particular example constitutes a bug is perhaps open to some interpretation. Using the standard unchecked Java exceptions to indicate pathological input values (as violations of preconditions) is recommended practice [10]. Ideally, in a real application, such use would be indicated in the Javadoc code documentation; the responsibility of checking for illegal input, and reporting it to the user, falls to any client that uses the method. A solution to such ambiguities in code could be to use an annotation framework, in addition to comments, to indicate the result of violating method preconditions. However, Artemis has not been written with such use in mind.

4.2 Analysis of Larger Programs Containing Known Errors

For a sanity check, we analysed some test programs, originally used for JCrasher [23], and subsequently re-evaluated by Tomb et al. [79]. The test classes are versions of the `P1` program, a homework assignment, and we used the submissions of five students, identified only by student number, and the contents of which are briefly summarised in Table 4.1.

During our first attempt [9], neither program s2120 nor program 3426 proceeded beyond intraprocedural analysis. As discussed in §3.2.3, we decided to switch to the Green framework [83]. After its adoption, we tested two of the available solvers, namely, CVC3 [8] and Z3 [62]. The initial small tests with CVC3 looked promising, but for the longer path conditions we aimed to handle, some of the constraints we passed on for solving crashed, neither Artemis

Table 4.2: Analysis of P1 for call depth 0.

	B	Paths	Warn	PC	Gen	Err	Time (ms)
s1	1	84	7	11	7	6	4262
	2	210	32	16	32	7	6012
	5	5654	881	31	881	7	47131
s1139	1	80	7	7	7	7	3924
	2	156	10	11	10	9	4640
	5	3107	106	26	106	9	14933
s2120	1	76	7	8	7	6	4544
	2	112	8	10	8	7	4649
	5	352	11	16	11	7	5690
s3426	1	93	16	6	12	9	4664
	2	166	27	10	18	10	5373
	5	726	60	22	36	10	9295
s8007	1	50	3	4	3	2	3649
	2	58	3	4	3	2	3703
	5	82	3	4	3	2	3969

B:	Branch bound
PC:	Maximum length path condition analysed
Gen:	Number of test cases generated
Err:	Number of unique errors triggered

nor Green, but the JVM itself. Although crashes were unexpected, that the JVM itself crashed was not entirely surprising, since CVC3 is called as a natively compiled library from Green. Trying to glean detail from the JVM profiler and tools like Valgrind did not yield any useful results, so we switched to Z3, which seemed to do the trick, and which we used for the rest of our experiments.

Results We analysed the five programs on Machine 2, and we compared our results, given in Tables 4.2 to 4.4, those of Tomb et al. [79]. For each of programs s1 and s1139, Artemis found an additional error, and for program s2120, we found two additional errors. However, for program s3426, we found one fewer `NullPointerException`. In summary:

s1 We found seven unique errors: Five `NullPointerException`s and two `ArrayIndexOutOfBoundsException`s.

s1139 We found nine unique errors: One `ArithmeticException`, one `NegativeArraySizeException`, five `NullPointerException`s, and finally, two `ArrayIndexOutOfBoundsException`s.

Table 4.3: Analysis of P1 for call depth 1.

	B	Paths	Warn	PC	Gen	Err	Time (ms)
s1	1	75	5	7	5	5	4220
	2	136	17	11	17	7	5189
	5	429	60	17	60	7	7267
s1139	1	90	7	7	7	7	4099
	2	176	10	11	10	9	4823
	5	3217	106	26	106	9	15736
s2120	1	97	9	8	8	6	4554
	2	214	10	10	9	7	5227
	5	11535	13	16	12	7	75830
s3426	1	98	16	6	12	9	4699
	2	211	27	10	18	10	5809
	5	2941	60	22	36	10	25067
s8007	1	50	3	4	3	2	3917
	2	58	3	4	3	2	3832
	5	82	3	4	3	2	4402

Table 4.4: Analysis of P1 for call depth 2.

	B	Paths	Warn	PC	Gen	Err	Time (ms)
s1	1	75	5	7	5	5	4093
	2	136	17	11	17	7	5071
	5	447	60	17	60	7	7583
s1139	1	92	7	7	7	7	4033
	2	186	10	11	10	9	4837
	5	3244	106	26	106	9	16330
s2120	1	99	9	8	8	6	4734
	2	246	10	10	9	7	5561
	5	11631	13	16	12	7	76914
s3426	1	98	16	6	12	9	4758
	2	211	27	10	18	10	5784
	5	2959	60	22	18	10	25485
s8007	1	52	3	4	3	2	3679
	2	70	3	4	3	2	3800
	5	115	3	4	3	2	4402

s2120 We found the same `ArrayIndexOutOfBoundsException` and seven `NullPointer Exceptions`, of which two are new.

s3426 We found the same `ArrayIndexOutOfBoundsException` and `NegativeArraySize Exception`, and like [79], we missed the `NumberFormatException`, known in [23]. We missed one `NullPointerException` found by [79].

s8007 We found two unique errors: One `NegativeArraySizeException` and one `Null PointerException`.

Discussion Artemis found four additional errors when compared to [79]. We speculate that this is the result of our branch counters being path-sensitive. They were not in [79], meaning every time an instruction is reached, no matter on what path, the bounds were enforced; so, some behaviours were artificially truncated, and therefore, not analysed for bugs.

Since setting the maximum call depth to zero is tantamount to performing intraprocedural analysis—implying all method calls effectively return unknown symbolic values—and considering we would expect more real errors to be uncovered the deeper the analysis goes, it was at first glance surprising to notice the decrease of the number of test cases generated, from seven to five, and the number of errors found, from six to five, for program s1 when the branch bound is kept one, but the call depth is increased from zero to one. However, upon closer examination of the generated tests and the original source code, we happened upon a subtle interplay between the branch bound and call depth: The offending statement, a call to a helper method, is inside a nested loop, the inner of which is guard by an `if` statement. Here, the branch bound prevented the call to the helper method from being reached; indeed, as soon as the branch bound was increased to two for the same call depth, the error re-emerged.

It is also interesting to note that there are apparently two distinct kind of behaviours programs can exhibit with increasing call depth: (1) Either the number of paths explored, and therefore, the number of warnings found, increase, like programs s1139 and s8007, or (2) they decrease and/or stay in the same range, like program s1. We reason that, in the former case, as deeper calls are allowed, more code is executed, and more warnings uncovered. In the latter case, it seems that the extra information following from the deeper calls actually translates into more infeasible constraints, where conjuncts added to the path condition in a called method contradict those added in a caller method.

The rather spectacular increase of the number of paths explored in program s2120, between call depth 0 and call depth 1, seems due to a particular method where, unlike the other students, this student favoured a recursive approach inside a complex nested loop. None of the programs contains calls deeper than two, so we did not expect an increase in bugs below call depth 1. The only way more bugs could be uncovered is if there exist concrete values larger than the branch bounds for which more bug-ridden paths would be followed.

We could not find an explanation for our missing one `NullPointerException` in program s3426. There were two unique error contexts for which warnings were flagged, but which did not result in test cases being generated. However, these possible errors concerned string concatenation to a string literal (which cannot be `null`), so we are least satisfied that they should not have resulted in test cases.

Bounding the Length of the Path Condition Since we planned to analyse a large program, we were interested what effect a bound on the length of the path condition would have. So, in a separate run, we set such a bound to be $10(c+1)$ for call depth c, and we stopped executing any path that reaches its bound. In all cases, the numbers of errors triggered correspond exactly to those of unbounded analysis.

When the path condition bound is in effect, but no path runs into the bound, the difference between times for bounded and unbounded runs (with respect to the path condition) is within 5%. Where it does trim paths, the savings were significant—up to 86% in the case of program s1 for a branch bound of five and a call depth of zero. With a bound on the length of the path condition in effect, we seem to be no worse off, with respect to running time and errors found, than without, and when there are savings with respect to running time, they are considerable.

Increasing the Branch Bound Increasing the branch bound had one interesting effect, which is not visible in Tables 4.2 to 4.4: For those programs that throw an `ArrayIndexOutOfBoundsException`, a larger branch bound resulted in a large number of test cases that triggered exactly the same error. In program s1139, going from branch bound two to branch bound five caused a fifty-fold increase in the number of successful test cases. We believe this is, essentially, because the larger branch bound allows more paths to be followed to array bound problems inside loops. Also, the way we handle indexing on symbolic arrays is to consider both possibilities, of indexing off the front (by a negative index) or indexing off the end (by and index greater than then array length). So, we expect some duplication in getting the same point of error.

4.3 Analysis of the Java PathFinder

To test Artemis on large code base, we ran the tool on the Java PathFinder. Of the 766 top-level classes for which we had bytecode at out disposal, we managed to analyse 702. The classes removed from consideration roughly falls into the following categories: (i) those JUnit setup and test files included in the code base under analysis (because their reflective interface crashed native code in the JVM), (ii) those that contain multi-dimensional arrays (only six classes), (iii) those that needed string handling to an extent we could not handle (only seven classes), and (iv) those that suffered from other issues. In the latter case, analysis and test runs simply got stuck for some concurrent and event-driven classes. We added a time bound

Table 4.5: Analysis of the Java PathFinder for various call depths and branch bounds.

C[a]	B[b]	Paths	Warn	OC[c]	Solved[d]	Gen[e]	Ran[f]	Err[g]	Time (h)
0	1	35409	8492	41.09%	6361	2270	1191	1081	1:20:38
0	2	43200	8970	40.54%	6764	2403	1212	1076	1:21:06
1	1	40841	9994	38.32%	7883	2775	1257	972	1:23:15
1	2	160795	25076	44.92%	22140	7152	1334	972	7:16:34
2	1	69267	17381	39.00%	15573	3779	1257	938	1:33:10
2	2	495294	117628	43.81%	115845	22603	1322	918	7:20:00
3	1	230832	62126	31.64%	58816	5439	1430	947	2:22:00
3	2	1837938	501070	34.08%	491226	10579	2995	914	16:01:47

[a]Call depth
[b]Branch bound
[c]Number of obvious contradictions
[d]Number of warnings for which solutions were available
[e]Number of test cases generated
[f]Number of test cases that successfully triggered an error
[g]Number of unique errors found

by annotation on the generated JUnit source, but to little effect. Some further exploration uncovered a possible explanation: It seems that Soot incorrectly (with respect to respect to JVM semantics) groups JVM bytecode instructions for handling exceptions into a single Soot statement. Although the grouping works in most cases, we discovered that bytecode generated by the standard Java compiler rely on these instructions being separate for exception propagation. The combined Soot statement resulted both in crashes and endless looping. Though we added a work-around, we are not sure that we covered all cases.

Results Our results for running Artemis on the Java PathFinder for call depths of 0, 1, 2, and 3, and for branch bounds of 1 and 2 are given in Table 4.5. To ensure that the runs completed in reasonable time, we applied bounds of 20, 20, 30, and 40, respectively and in this order, to the length of the path conditions of call depths 0 to 3. Also, Artemis was allowed to analyse private methods at call depth 0, to establish a baseline for the number of errors; this behaviour was turned off at deeper levels. When allowed to progress to a call depth of 3 with a branch bound of 2, Artemis found 914 unique bugs in just over 16 hours. Here, we determined uniqueness by exception type and line number.

Discussion With a branch bound of 1, we expect to reach all errors that do not depend on particular concrete values. When we consider the drop in the number of unique errors found, from call depth 0 to call depth 1 and below, it seems that our baseline overapproximated the number of real errors in the end by about 10%.

For call depth 1, branch bound 2, we could only solve for about 79% of the warnings; for call depth 3, branch bound 2, this rose to 98%. (We don't consider call depth 0 here, because there Artemis was allowed to execute private methods as well.) We speculate that, although more paths are explored the deeper we go, with a larger branch bound, the infeasible paths are pruned relatively early, and that the remaining paths actually keep going back to the same set of errors. This is also motivated by 77% of successfully triggered errors being unique for call depth 1, branch bound 1, but only 30% of those for call depth 3, branch bound 2.

Of particular note is that the number of paths pruned because of an obvious contradiction —where the symbolic execution engine, when adding a constraint q, first checks for $\neg q$ in the path condition—fall in the 30% to 45% range. Therefore, a significant number of infeasible paths are actually pruned before being sent to the solver, which is a comparatively expensive operation.

That only about 2% of the solutions translate into successfully generated test cases on call depth 3, branch bound 2 seemed troubling at first. However, a closer examination revealed that 94% of these solutions were in fact eliminated before code generation, because their associated test cases would have entailed assignment to a `final` field.

Bounding the Length of the Path Condition For each of the call depths, at branch bound 1, we also ran the analysis with the bounds on the path condition turned off. This resulted in a significant increase in processing time the deeper we went, but with very small gains in terms of unique bugs found—in the order of 1%. Indeed, for call depth 3, branch bound 1, analysis without a bound on the length of the path condition found seven extra bugs, but at the expense of nine extra hours of processing, about 460% of the original time. When keeping to intraprocedural analysis on branch bound 2, removing the limit on the path condition length resulted in nine extra bugs, but at 330% of the original time.

Without path condition length bounds, at call depths 0 to 3 for branch bound 1, only 16, 34, 32, and 21 path conditions, respectively, were longer than the maximum path lengths we associated with the bounded runs at these levels. So, in any case, there seems to be few very long paths that lead to errors, and these paths dominate the running time when path conditions of arbitrary length are allowed. Therefore, we elected to enforce the path length bounds during our main analysis.

Cyclomatic Complexity and Errors The most error-prone method, according to our analysis at call depth 3 and branch bound 2, contained six unique errors. A further one contained five errors, 39 contained between two and four errors, with the remaining methods with errors contained one error each. Surprisingly, in the light of the common wisdom that routines with high cyclomatic complexity are particularly error-prone, only two of the methods with three to six errors had a cyclomatic complexity of thirteen and fifteen, which the UCC tool flagged as in the middle of "medium" complexity; the others were rated at eight and lower.

Conversely, we looked at the methods analysed with the highest cyclomatic complexity. There are 11 methods with high cyclomatic complexity in the code base. We analysed five of these. Four of the analysed methods operate over primitive values only, without integer division, so none of the errors we cover are possible; the fifth, although with opportunity for error, was presumably pruned somewhere. The data supports no final conclusions on this matter. We are left to muse, somewhat truistically, that a method does not have to be particularly complex to contain relatively many errors.

Chapter five

Conclusion and Future Work

In this dissertation, we introduced the Artemis tool for finding run-time errors in Java programs. The core feature of the tool is that it not only finds the potential for errors in a goal-directed way by symbolic execution, but it can also confirm that the expected errors exist by producing test cases to show the error occurring. We showed how effective it is in finding errors in some small-sized examples, where some of the errors were missed by previous analyses of the same code. We concluded our results with the analysis of a larger program, namely, the Java PathFinder, and we showed we could successfully trigger a number of different errors.

Our contributions to the field of bug-finding are:

1. the externalisation of testing through the generation and running of test cases in a standard unit testing framework,
2. the application of revisitation bounds in a path-sensitive way, and
3. the handling of run-time exceptions thrown explicitly by the code under review.

We also took some tentative steps in examining the effects of parameters like call depth, branch bounds, and bounds on the length of the path condition on the analysis of a large and complex code base.

5.1 Future Work

A number of extensions to our work is evident, and some issues remain that are not fully explored. We summarise these in the following paragraphs.

Constraint Solving The Green framework that we used as front end to constraint solving contains two important features we did not use: Persistent caching of computed solutions, and slicing to enable the re-use of previous results. We already get good results by pruning on obvious contradictions in the symbolic execution engine, and we expect that using the full abilities of Green might be particularly interesting in conjunction with our existing results.

Also, we need to explore the possibilities of other constraint solvers. Our current constraint solver of choice, Z3, has some limitations, for example, with respect to linearity, and it should be interesting to see if we can make our approximation of error-locating constraints more precise.

Threading Currently, Artemis runs in a single thread. However, threading is possible since (i) we assume methods have no side-effects, and (ii) once symbolic execution has determined that a possible error has been found, the constraint solving and test case generation can be handled independently from the continued symbolic execution. A threaded solution with a pool of available workers should make more efficient use of multi-processor, multi-core machines.

Test-Case Generation Currently, setting up object state for test case generation is a somewhat ad hoc affair. Artemis already collects information on constructors, to determine which is the way of least resistance when writing object instantiation statements for the test source code. It also contains the (unused) ability to collect the return states for the constructors of the classes under analysis.

Java language semantics provide three main ways of instantiating objects: (i) with a constructor (even if via a factory method), (ii) by a clone operation, and (iii) by deserialising a serialised object form. Of these, the second is particularly prone to misuse [10], and the last has a lot of technical, I/O-related details, especially from the view of our approximate analysis. A simple, brute-force approach on the first might yield some useful results: Initiate analysis of a method with return states, each in turn, collected earlier from all constructors of a class, and try to keep the accompanying explosion of paths in check.

A different, possibly more efficient approach would be to attempt a tie-up between unused solutions from the constraint solver—implying that they originated not through an argument passed into a method, but through some field access—and the unknown values in the collected return states of constructors. Although this would not solve the problem of where a sequence of top-level method calls results in an error state, it might catch errors resulting from constructors that do not enforce preconditions for object state or do so carelessly.

Extend the Collection of Code Under Analysis Currently, Artemis does, for example, analyse classes in the standard Java libraries. Also, the classes unanalysed in our current setup should be examined and handled, as well as other large examples considered. Some of these might possibly be handled better by exploiting the abilities of Green to the fullest.

Add Missing Functionality Artemis has incomplete support for multi-dimensional arrays, which we need to remedy. There are also some issues, originating in Soot, with the handling of the standard exceptions checked for concurrent code, which need addressing.

With hindsight, it might not be amiss to reevaluate our design and approach from scratch. In the end, Soot caused a lot of trouble, and its elimination may make our analysis run smoother. Also, with the availability of Green, tighter integration with its set of operators and expressions might result in a faster and more robust symbolic execution engine.

5.2 The Big Picture

In the previous section, we remarked on how Artemis can be improved on a technical level. Its ultimate goal, however, should be assisting the programmer during programming, and not afterwards as it was presented here [85]. We believe a dramatic improvement in code quality is possible if errors are highlighted as they are introduced. Although this already happens to some extent in automatic static analyses performed by IDEs such as Eclipse, our tool could be useful in illuminating errors not handled by classic static analysis.

For example, used as part of an IDE, the tool will always be handling relatively small pieces of code, which would immediately alleviate the problem of scalability. Furthermore, although our examples didn't contain any false positives, it is still possible that our technique can produce errors that are spurious since there might be hidden preconditions for methods not visible at the level of the code; if you analyse code as it is being generated though, the developers will be forced to make such preconditions visible through code contracts (or by conditions in the code) which will immediately lead to better code. In this way one can derive preconditions on-the-fly during programming.

Finally, we collect a large number of metrics during this analysis and we believe this can also be used to derive code complexity metrics as well as code quality metrics. Here we simply presented the error-detection, but of course the number of errors reported by the tool can also be used to measure the quality of the code. Furthermore, the number of paths analysed or the size of the maximum path condition (which would be a similar measure to cyclomatic complexity) can similarly be used to measure code complexity.

Bibliography

[1] The Java PathFinder. `http://babelfish.arc.nasa.gov/trac/jpf`. [35, 61]

[2] The JUnit testing framework. `http://www.junit.org/`. [10, 43]

[3] Unified code count. `http://csse.usc.edu/csse/affiliate/private/codecount.html`. [62]

[4] Abramsky, S., and Hankin, C. An introduction to abstract interpretation. In *Abstract Interpretation of Declarative Languages* (1987), vol. 1, Ellis Horwood, pp. 63–102. [1, 25, 26]

[5] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, & Tools*, second ed. Pearson, Boston, Mass., 2007. [24, 25]

[6] Anand, S., Păsăreanu, C. S., and Visser, W. Symbolic execution with abstract subsumption checking. In *Model Checking Software*, A. Valmari, Ed., vol. 3925 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 163–181. [34]

[7] Babbage, C. *Passages from the Life of a Philosopher*. Longman, Green, Longman, Roberts, & Green, 1864. [13]

[8] Barrett, C., and Tinelli, C. CVC3. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Heidelberg, 2007), CAV'07, Springer-Verlag, pp. 298–302. [10, 67]

[9] Bester, W. H. K., Inggs, C. P., and Visser, W. C. Test-case generation and bug-finding through symbolic execution. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference* (New York, NY, USA, 2012), SAICSIT '12, ACM, pp. 1–9. [ix, xi, 61, 67]

[10] Bloch, J. *Effective Java*, second ed. Addison-Wesley, Boston, Mass., 2008. [39, 43, 58, 67, 76]

[11] Boehm, B. W. Guidelines for verifying and validating software requirements and design specifications. In *Proceedings of the European Conference on Applied Information Technology (IFIP'79)* (1979), pp. 711–719. [15]

[12] Bourque, P., and Dupuis, R., Eds. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, Los Alamitos, CA, 2004. [14]

[13] Boyer, R. S., Elspas, B., and Levitt, K. N. SELECT—a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software* (New York, NY, USA, 1975), ACM, pp. 234–245. [33]

[14] Cadar, C., Dunbar, D., and Engler, D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 209–224. [34]

[15] Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., and Engler, D. R. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security 12*, 2 (Dec. 2008), 10:1–10:38. [34]

[16] Clarke, E. The birth of model checking. In *25 Years of Model Checking*, O. Grumberg and H. Veith, Eds., vol. 5000 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 1–26. [24, 26, 27]

[17] Clarke, E. M., and Emerson, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, D. Kozen, Ed., vol. 131 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1982, pp. 52–71. [27]

[18] Clarke, L. A. A system to generate test data and symbolically execute programs. *Software Engineering, IEEE Transactions on SE-2*, 3 (sept. 1976), 215–222. [2, 30]

[19] Collier, B. *The little engines that could've: The calculating machines of Charles Babbage*. PhD thesis, Harvard University, 1970. [13]

[20] Consortium for IT Software quality. CISQ executive forums. Carnegie Mellon Software Engineering Institute, 2009. [14]

[21] Cousot, P. Abstract interpretation based formal methods and future challenges (electronic version). In *Informatics*, R. Wilhelm, Ed., vol. 2000 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2001, pp. 138–156. [16, 26]

[22] Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1977), POPL '77, ACM, pp. 238–252. [24, 25]

[23] Csallner, C., and Smaragdakis, Y. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience 34*, 11 (Sept. 2004), 1025–1050. [34, 61, 67, 70]

[24] Csallner, C., and Smaragdakis, Y. Check'n'crash: combining static checking and testing. In *Proceedings of the 27th International Conference on Software Engineering* (2005), ACM, pp. 422–431. [34]

[25] Csallner, C., Smaragdakis, Y., and Xie, T. DSD-Crasher: a hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology (TOSEM) 17*, 2 (2008), 8. [34]

[26] De Bakker, J. W. *Mathematical Theory of Program Correctness*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1980. [32]

[27] Dijkstra, E. W. *A Discipline of Programming*. Series in Automatic Computing. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976. [21, 23]

[28] Dillon, L. Using symbolic execution for verification of Ada tasking programs. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 4 (1990), 643–669. [34]

[29] Elspas, B., Levitt, K. N., Waldinger, R. J., and Waksman, A. An assessment of techniques for proving program correctness. *ACM Comput. Surv. 4*, 2 (June 1972), 97–147. [17]

[30] Emerson, E. The beginning of model checking: A personal perspective. In *25 Years of Model Checking*, O. Grumberg and H. Veith, Eds., vol. 5000 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 27–45. [14, 15, 17, 18, 24, 27]

[31] Floyd, R. W. Assigning meaning to programs. *Mathematical Aspects of Computer Science 19* (1967), 19–31. [18, 22]

[32] Foley, J., and Murphy, C. Q&A: Bill Gates on trustworthy computing. *Information Week* (May 2002). [14]

[33] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995. [39]

[34] Girgis, M. An experimental evaluation of a symbolic execution system. *Software Engineering Journal 7*, 4 (1992), 285–290. [34]

[35] Godefroid, P., Klarlund, N., and Sen, K. Dart: directed automated random testing. In *Programming Language Design and Implementation* (2005), pp. 213–223. [34]

[36] Goldstine, H. H., and Von Neumann, J. *Planning and Coding of Problems for an Electronic Computing Instrument*, vol. II, Part II. Institute for Advanced Study, Princeton, New Jersey, 1948. [17]

[37] Gosling, J., Joy, B., Steele, G., and Bracha, G. *The Java Language Specification*, third ed. Addison-Wesley, Boston, MA, 2005. [1, 38]

[38] Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969), 576–580. [14, 21, 23]

[39] Huth, M., and Ryan, M. *Logic in Computer Science: Modelling and Reasoning about Systems*, second ed. Cambridge University Press, Cambridge, UK, 2004. [21, 27]

[40] IEEE Std 610. IEEE standard glossary of software engineering terminology, September 1990. [16]

[41] Jussien, N., Rochart, G., and Lorca, X. The CHOCO constraint programming solver. In *CPAIOR08 workshop on OpenSource Software for Integer and Contraint Programming OSSICP08* (2008), pp. 1–10. [10, 42]

[42] Kemmerer, R., and Eckmann, S. Unisex: A Unix-based symbolic executor for Pascal. *Software: Practice and Experience 15*, 5 (1985), 439–458. [34]

[43] Khurshid, S., Păsăreanu, C., and Visser, W. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, H. Garavel and J. Hatcliff, Eds., vol. 2619 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003, pp. 553–568. [35]

[44] Kildall, G. A. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1973), POPL '73, ACM, pp. 194–206. [25]

[45] King, J. C. A new approach to program testing. In *Proceedings of the International Conference on Reliable Software* (New York, NY, USA, 1975), ACM, pp. 228–233. [2, 30, 37]

[46] King, J. C. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (July 1976), 385–394. [2, 29, 33, 37, 50]

[47] Korel, B. Automated test data generation for programs with procedures. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software testing and analysis* (New York, NY, USA, 1996), ISSTA '96, ACM, pp. 209–215. [34]

[48] Korel, B., and Laski, J. Dynamic program slicing. *Information Processing Letters 29*, 3 (1988), 155–163. [29]

[49] Kurshan, R. Verification technology transfer. In *25 Years of Model Checking*, O. Grumberg and H. Veith, Eds., vol. 5000 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 46–64. [15]

[50] Laski, J., and Stanley, W. *Software Verification and Analysis*. Springer-Verlag, London, UK, 2009. [14, 16, 24, 25, 28, 30]

[51] Lewis, J., and Chase, J. *Java Software Structures: Designing and Using Data Structures*, third ed. Addison-Wesley, 2009. [14]

[52] Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification*, second ed. Prentice Hall, Upper Saddle River, NJ, 1999. [1, 39, 40, 41]

[53] London, R. L. The current state of proving programs correct. In *Proceedings of the ACM Annual Conference—Volume 1* (New York, NY, USA, 1972), ACM '72, ACM, pp. 39–46. [19]

[54] Majumdar, R., and Sen, K. Hybrid concolic testing. In *International Conference on Software Engineering* (2007), IEEE Computer Society, pp. 416–426. [34]

[55] Mattis, M. Repurposing Ada. Salon.com web article, `http://www.salon.com/1999/03/16/feature_217/`, March 1999. [13]

[56] McCabe, T. A complexity measure. *IEEE Transactions on Software Engineering 2*, 4 (1976), 308–320. [62]

[57] McCarthy, J. A basis for a mathematical theory of computation, preliminary report. In *Papers presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference* (New York, NY, USA, 1961), IRE-AIEE-ACM '61 (Western), ACM, pp. 225–238. [17]

[58] McCarthy, J. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, P. Brafford and D. Hirshberg, Eds. North-Holland, 1963, pp. 33–70. [17]

[59] Metsker, S. J., and Wake, W. C. *Design Patterns in Java*, second ed. Addison-Wesley, Boston, Mass., 2006. [39]

[60] Morris, F. L., and Jones, C. B. An early program proof by Alan Turing. *IEEE Ann. Hist. Comput. 6*, 2 (Apr. 1984), 139–143. [17]

[61] Morrison, G. C., Inggs, C. P., and Visser, W. C. Automated coverage calculation and test case generation. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference* (New York, NY, USA, 2012), SAICSIT '12, ACM, pp. 84–93. [34]

[62] Moura, L., and Bjørner, N. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. Ramakrishnan and J. Rehof, Eds., vol. 4963 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 337–340. [10, 67]

[63] Naur, P. Proof of algorithms by general snapshots. *BIT Numerical Mathematics 6* (1966), 310–316. [17]

[64] Parr, T. The StringTemplate template engine. `http://www.stringtemplate.org/`. [10]

[65] Pnueli, A. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, 1977* (31 1977-nov. 2 1977), pp. 46 –57. [27]

[66] Prior, A. N. *Past, present and future.* Oxford University Press, 1967. [27]

[67] Păsăreanu, C., and Visser, W. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer (STTT) 11* (2009), 339–353. 10.1007/s10009-009-0118-1. [26, 29, 34]

[68] Păsăreanu, C. S., and Rungta, N. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2010), ASE '10, ACM, pp. 179–180. [35]

[69] Schiller, T. W., and Ernst, M. D. Rethinking the economics of software engineering. In *Proceedings of the FSE/SDP workshop on Future of software engineering research* (New York, NY, USA, 2010), FoSER '10, ACM, pp. 325–330. [14]

[70] Schwartzbach, M. I. Lecture notes on static analysis. BRICS, Department of Computer Science, University of Aarhus, Denmark. [24, 25]

[71] Sedgewick, R., and Wayne, K. *Introduction to Programming in Java: An interdisciplinary Approach.* Addison-Wesley, Boston, Mass., 2008. [41, 67]

[72] Sen, K., and Agha, G. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification* (2006), pp. 419–423. [34]

[73] Sen, K., Marinov, D., and Agha, G. CUTE: a concolic unit testing engine for C. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2005), pp. 263–272. [34]

[74] Shannon, D., Hajra, S., Lee, A., Zhan, D., and Khurshid, S. Abstracting symbolic execution with string analysis. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007* (2007), pp. 13–22. [34]

[75] Smith, B. Linux: Show the number of CPU cores and sockets on your system. IBM Developer-Works blog, April 2012. [61]

[76] Sălcianu, A. Notes on abstract interpretation. Unpublished manuscript, November 2001. [25]

[77] Tassey, G. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007.011 (2002). [1, 14]

[78] Tillmann, N., and De Halleux, J. Pex: white box test generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs* (Berlin, Heidelberg, 2008), TAP'08, Springer-Verlag, pp. 134–153. [34]

[79] Tomb, A., Brat, G., and Visser, W. Variably interprocedural program analysis for runtime error detection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2007), ISSTA '07, ACM, pp. 97–107. [8, 37, 52, 61, 67, 68, 70]

[80] Turing, A. M. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines* (June 1949), University Mathematical Laboratory, Cambridge. [17]

[81] Turing, A. M. On computable numbers, with an application to the Entscheidungsproblem. In *The Essential Turing: The ideas that gave birth to the computer age*, B. J. Copeland, Ed. Oxford University Press, Oxford, UK, 2004, ch. 1, pp. 1–90. [15]

[82] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. Soot – a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (1999), CASCON '99, IBM Press, pp. 13–. [6, 7, 41]

[83] Visser, W., Geldenhuys, J., and Dwyer, M. B. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2012), FSE '12, ACM, pp. 58:1–58:11. [10, 42, 67]

[84] Visser, W., Păsăreanu, C. S., and Pelánek, R. Test input generation for Java containers using state matching. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2006), ISSTA '06, ACM, pp. 37–48. [34]

[85] Whalen, M. W., Godefroid, P., Mariani, L., Polini, A., Tillmann, N., and Visser, W. FITE: future integrated testing environment. In *FSE/SDP Workshop on the Future of Software Engineering Research* (2010), G.-C. Roman and K. J. Sullivan, Eds., ACM, pp. 401–406. [77]

[86] Whitehead, A. N., and Russell, B. *Principia Mathematica*. Cambridge University Press, Cambridge, UK, 1910–1913. [15]

[87] Wirth, N. *Systematic Programming: An Introduction*. Series in Automatic Computing. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973. [22, 29]

[88] Xu, R.-G., Godefroid, P., and Majumdar, R. Testing for buffer overflows with length abstraction. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2008), ISSTA '08, ACM, pp. 27–38. [34]

[89] Zhang, J. Symbolic execution of program paths involving pointer and structure variables. In *Proceedings of the Quality Software, Fourth International Conference* (Washington, DC, USA, 2004), QSIC '04, IEEE Computer Society, pp. 87–92. [34]