Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science

Kari Kähkönen

# Automated Dynamic Test Generation for Sequential Java Programs

Master's thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Technology

Espoo, 31th July 2008

Supervisor: Prof. Ilkka Niemelä
Instructor: Docent Keijo Heljanko

| | |
|---|---|
| HELSINKI UNIVERSITY OF TECHNOLOGY<br>Faculty of Information and Natural Sciences<br>Degree Programme of Computer Science and Engineering | ABSTRACT OF MASTER'S THESIS |
| Author: Kari Kähkönen | Date: 31th July 2008<br>Pages: viii + 58 |
| Title of thesis: Automated Dynamic Test Generation for Sequential Java Programs | |
| Professorship: Theoretical Computer Science | Code: T-119 |
| Supervisor: Prof. Ilkka Niemelä | |
| Instructor: Docent Keijo Heljanko | |

Testing in software industry has traditionally been done by using manually generated test cases. Given the high cost of this approach, there is interest to create methods that allow test cases to be generated automatically. This work studies how dynamic symbolic execution can be used to generate test cases for sequential Java programs in an automated fashion.

The main method employed in this work is dynamic symbolic execution, where a program is executed both concretely and symbolically at the same time. During an execution a set of symbolic constraints describing the input values that will force the program to follow an unexplored execution path is collected. By solving the collected constraints new input values are obtained allowing each test run to exercise different behaviour of the program.

Combining symbolic execution with concrete execution can be done with code instrumentation. By keeping track of the constraints generated during all previous test runs it is possible to use different strategies to choose the execution path that will be tested next. The way execution paths are chosen becomes increasingly important with programs that have a large enough number of execution paths so that it is infeasible to explore them all. In this work a testing framework is developed that allows multiple test runs to be executed concurrently during the symbolic execution process. This allows the utilisation of multicore processors and networks of computers in test generation. Another contributions in this work are the description of how input objects can be initialised lazily and the introduction of a technique that can avoid generating some unnecessary test cases when objects are used as inputs.

Based on the methods developed in this work, a tool has been implemented that can be used for automated testing of sequential Java programs. The implemented approach is evaluated through case studies and the scalability of dynamic symbolic execution for large programs is discussed.

Keywords: Dynamic symbolic execution, test case generation, code instrumentation, constraint solving, testing Java programs

| | |
|---|---|
| TEKNILLINEN KORKEAKOULU<br>Informaatio- ja luonnontieteiden tiedekunta<br>Tietotekniikan koulutusohjelma | DIPLOMITYÖN<br>TIIVISTELMÄ |
| Tekijä: Kari Kähkönen | Päiväys: 31. heinäkuuta 2008 |
| | Sivumäärä: viii + 58 |
| Työn nimi: Automatisoitu dynaaminen testien generointi Java-ohjelmille | |
| Professuuri: Tietojenkäsittelyteoria | Koodi: T-119 |
| Työn valvoja: Prof. Ilkka Niemelä | |
| Työn ohjaaja: Dosentti Keijo Heljanko | |

Testausta ohjelmistoteollisuudessa on perinteisesti tehty ihmistyönä luotujen testitapausten avulla. Koska kyseinen menetelmä on kallis sen vaatiman työmäärän johdosta, automaattisille menetelmille testitapausten tuottamiseen on tarvetta. Tässä työssä tarkastellaan, kuinka dynaamista symbolista suoritusta voidaan käyttää testitapausten tuottamiseen automatisoidusti Javalla toteutetuille ohjemille, joissa ei esiinny rinnakkaisuutta.

Työssä käytetty dynaaminen symbolinen suoritus on menetelmä, jossa testattava ohjelma suoritetaan sekä konkreettisesti että symbolisesti yhtä aikaa. Suorituksen aikana menetelmässä kerätään symbolisia rajoitteita kuvaamaan syötearvojen joukkoja, jotka pakottavat ohjelman suorituksen seuraamaan aikaisemmin tutkimatonta suorituspolkua. Ratkaisemalla kerätyt rajoitteet saadaan muodostettua uusia konkreettisia syötearvoja ja näin ollen jokainen testisuoritus saadaan testaamaan toisista testiajoista poikkeavaa käyttäytymistä.

Symbolisen suorituksen yhdistäminen konkreettisilla arvoilla suorittamiseen on mahdollista toteuttaa instrumentoimalla annetun ohjelman suoritettavaa versiota. Pitämällä muistissa testiajojen aikana muodostetut rajoitteet voidaan seuraavan tutkimattoman suorituspolun valinta tehdä usealla erilaisella hakustrategialla. Menetelmällä, kuinka seuraava tutkittava polku valitaan, on kasvava merkitys silloin, kun testattavan ohjelman suorituspolkujen avaruus on liian suuri, jotta se voitaisiin käydä kattavasti läpi. Työssä kuvataan myös testausjärjestelmän rakenne, joka mahdollistaa useiden dynaamiseen symboliseen suoritukseen liittyvien testausajojen suorittamisen samanaikaisesti. Tämä mahdollistaa moniydinsuorittimien sekä useiden erillisten tietokoneiden käytön testitapausten generoimisessa. Muita tämän työn tuloksia ovat menetelmä alustaa syötteenä annettavia olioita laiskasti sekä tekniikka, jolla voidaan välttää turhien testien luominen tietyissä tapauksissa, kun olioita käytetään ohjelmaan syötteenä.

Työssä kehitettyihin menetelmiin pohjautuen on toteutettu työkalu, jota voidaan käyttää Java-ohjelmien testaukseen automatisoidusti. Toteutusta tarkastellaan tapaustutkimuksien pohjalta ja työstä saatujen kokemusten pohjalta dynaamisen symbolisen suorituksen skaalattavuutta laajoille ohjelmille arvioidaan.

Avainsanat

Dynaaminen symbolinen suoritus, testitapausten generoiminen, koodin instrumentointi, rajoitteiden ratkaiseminen, Java-ohjelmien testaus

# Acknowledgements

This thesis was done in the Department of Information and Computer Science at Helsinki University of Technology as a part of the research project LIME (LightweIght formal Methods for distributed component-based Embedded systems). I wish to thank my supervisor Professor Ilkka Niemelä for giving me the opportunity to work on this topic. I am also grateful to my instructor Docent Keijo Heljanko for his advice and invaluable comments. The financial support of Tekes, Conformiq Software Oy, Elektrobit Oy, Nokia Oyj and Space Systems Finland for funding in the LIME project is gratefully acknowledged as well as the funding from Technology Industries of Finland Centennial Foundation prior to LIME.

Finally, I wish to thank my parents and my beloved Tuulia for all the support they have given me.

Kari Kähkönen
Otaniemi, 31th July, 2008

# Contents

# List of Figures

# List of Tables

# List of Symbols and Abbreviations

| | |
|---|---|
| $\wedge$ | logical conjunction |
| $\vee$ | logical disjunction |
| $\Rightarrow$ | logical implication |
| $\mapsto$ | mapping |
| $\in$ | element of |
| $\notin$ | not an element of |
| $\cup$ | set union |
| $\cap$ | set intersection |
| $\emptyset$ | the empty set |
| $\top$ | true |
| $\mathcal{I}$ | mapping from input sequence numbers to input values |
| $\mathcal{S}$ | mapping from primitive type variables to symbolic values |
| $\mathcal{R}$ | mapping from object references to symbolic values |
| $\mathcal{M}$ | mapping from logical addresses to object references |
| BDD | binary decision diagram |
| BFS | breadth-first search |
| CFG | control flow graph |
| DFS | depth-first search |
| JDK | Java Development Kit |
| JVM | Java Virtual Machine |
| PC | path constraint |
| SMT | Satisfiability Modulo Theories |
| TCP | Transmission Control Protocol |

# Chapter 1

# Introduction

Ensuring that a software system or component works as intended is an important challenge for the software industry. Faulty software in safety critical systems, for example, in medical care can lead to the loss of human life and in commercial products faults can cause big financial losses. Two methods that are commonly used for checking correctness of software are testing and model checking [9]. In testing the actual system is used in verification by making test runs in order to determine whether the system works correctly in each individual test case. Traditionally the test cases have been generated manually which is often slow, error-prone and the adds up to a large part of the overall cost of the development process. The need for automated verification methods therefore clearly exists. In model checking, on the other hand, a model of a system is created with the aim to exhaustively test every possible execution in the model. The advantage with model checking is that it is possible to verify that the model of the system functions correctly in all possible situations where as with testing it is only possible to show that incorrect behaviour exists but not, in general, prove that the program is correct. Additionally model checking can be automated when the model and the properties to be checked are given. The drawback is that the state spaces of the models are ofter so huge that exploring all of the states is virtually impossible.

In this work the focus is on testing concrete implementations of systems. However, the aim is to develop automated methods that systematically generate test cases that cover majority of the behaviour of a system under test. Automated methods help in reducing the cost of testing and systematically generated test cases can provide the tester a good confidence that a system will function correctly. It can be seen that the testing methods developed in this work take a step towards model checking.

There are several test case generation methods that have been suggested over the years. One of the simplest is random testing [5], in which a number of inputs to the system are generated randomly. The system under test is executed with these inputs and it is checked that the test runs do not violate a given specification or that the test runs exercise enough of the intended behaviour of the system. Random testing is a lightweight method as it is easy to generate random inputs and a test run does not require any time or memory resources in addition to those needed by the execution of the program. However, random testing has its limitations. It might generate inputs that exercise the same behaviour multiple times and it is possible that to check some behaviour, very specific inputs would need to be generated, making it highly unlikely to get these inputs by random means in a reasonable time. Additionally, in random testing it is difficult to determine when the testing should be stopped as it is not known at any point whether the state space of a program has been fully explored.

Symbolic execution [22, 10, 21] is one proposed solution that addresses the limitations of random testing. The main idea in symbolic execution is to analyse a program so that it is possible to generate test inputs that will exercise different behaviour in each test run. The analysis is done by executing the given program symbolically, that is, using symbolic values in place of concrete ones in program execution. The symbolic values represent a set of possible concrete input values that will cause the execution to follow the current execution path. At each branching statement a condition is formed that constrains the set of input values that will force the execution to take the desired branch. The idea of symbolic execution is not a new one as it has been around from the 1970s, but the recent advancements in constraint solvers and the continuing improvement in modern computers have made the approach interesting as it is not anymore limited to only the simplest of programs.

There are two general approaches that have been suggested for symbolic execution: one is based on static analysis and the other on dynamic analysis. In static analysis [4] the given program is analysed by simulating it algorithmically. In other words, the analysis is done without executing the program. This is done by building a model of a given program and exploring how different executions change the state of this model. However, as already mentioned the state space of real life programs is generally so large that it is not reasonable to explore every reachable state of the system. This usually means that that an abstraction of the state space of the program is created and this abstract model is inspected. Usually the abstractions used are conservative (*i.e.*, all the reachable abstract states are guaranteed to contain all the reachable concrete states but some non-reachable concrete states might also get labelled as reachable) causing the methods to report spurious errors. Model checking is one example of a technique that falls under static analysis. In fact, it is one of the most accurate and therefore computationally expensive techniques at that category. Pure symbolic execution can also be seen as a static method as the inputs to the system are expressed and manipulated only in an abstract symbolic form.

Dynamic analysis, on the other hand, is based on the information collected during an actual execution of a given program. To be able to make the required observations at runtime, new lines of code are instrumented to the program or the program is executed in a virtual environment. The advantage of dynamic analysis over static analysis is that accurate information about the program state that might not be easily accessible in the static case is now available due to the concrete execution. On the downside, with dynamic analysis there is generally no possibility for similar conservative approximations like in the case of static analysis [16]. This has the implication that on large programs it is unlikely that all possible executions of the program can be analysed and thus we are limited only to testing the program instead of fully verifying it. In symbolic execution that is based on dynamic analysis, the program is executed both concretely and symbolically at the same time and the collected symbolic constraints are used to guide the concrete execution. For a general overview of the differences and similarities of static and dynamic analysis as well as discussion of combination of the two approaches, the reader is referred to [12]. Note also that symbolic execution is not limited only for test generation. For example, in [20] symbolic execution is applied for generation of likely invariants for data structures. However, in this work only the problem of test case generation is considered.

The goal of this work is to develop a test generation tool based on dynamic symbolic execution such that it can be used for testing of sequential Java programs. Our aim is also to identify the strong and weak points of the chosen approach to see where further research is needed. Other similar tools [27, 8, 17, 29, 18] have been developed but none of these are currently available with full open source code and licensing that allows modification of the tool with the exception of a recently released early version of CREST [18] that is a dynamic symbolic execution tool for C programs. Because the plan is to extend our tool to also handle concurrency in the future, Java was chosen as a target language as it has been designed to support concurrent programming. To the best of our knowledge there are only two tools based on or supporting symbolic execution that can be used to test concurrent programs, Java PathFinder [19, 3] with a symbolic execution extension (based on static analysis) and jCUTE [28] (based on dynamic analysis).

The rest of this work is structured as follows. In Chapter 2 the concepts needed for understanding symbolic execution are introduced and an overview is given on how the symbolic execution can be combined with concrete execution. Chapter 3 describes the instrumentation of the original program that makes symbolic execution possible. In Chapter 4 it is discussed how the information collected during a run of an instrumented program can be used to find new input values that will guide the concrete execution to cover behaviour that has not yet been tested. Chapter 5 covers some implementation details of our testing tool and in Chapter 6 some case studies are presented and their results discussed. Chapter 7 concludes the work and discusses topics for further research.

# Chapter 2

# Overview

The purpose of this chapter is to familiarise the reader with the used terminology and give an overview of how the developed test generation system works in general. The key concepts behind symbolic execution will be introduced first and then it will be explained through running examples how executing a program both concretely and symbolically can be used to generate test inputs that will explore distinct executions of the system under test.

## 2.1 Basic Concepts

A program written with an imperative programming language can be seen as consisting of a sequence of *statements* that are the smallest elements in a program that can be executed separately. By executing a statement a program can change its state. An *execution path* of a program $P$ is a sequence of statements that could be executed in the given order from the beginning of $P$. A *prefix* of length $n$ of an execution path $\pi$ is a sequence that consists of the first $n$ statements of $\pi$. For sequential programs, the execution path that will be followed is determined only by the input values of the program. Every value the program reads that is not decided solely by the execution history can be seen as an input (*e.g.*, values received from the user and the use of random value generators). Naturally, if multi-threaded programs are also considered, the thread scheduling introduces another source of non-determinism in addition to the one caused by input values. However, in this work the discussion is limited to sequential programs only.

**Example 1.** Figure 2.1 shows a simple Java program on the left and a control flow graph of the program on the right side. In this case, all of the paths from the start node to the end node in the control flow graph represent a potential execution path. If at the first line of the program an input value is read such that $x = -10$ and the program statements are represented by their line numbers, the resulting execution path is $(1, 2, 4, 6, 7, 9)$. In fact, any input value for the variable $x$ that is less than or equal to -5 will cause the program to follow the same execution path. It is also worth noting that not all of the possible paths in the control flow graph are actual

```
1 x = input();
2 x = x + 5;
3
4 if (x > 0)
5       y = input();
6 else
7       y = 10;
8
9 if (x > 2)
10      if (y == 2789)
11          error();
```

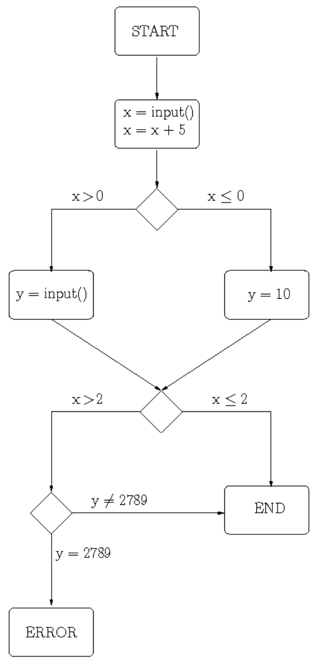


**Figure 2.1:** An example program and its control flow graph

execution paths. It is, for example, impossible to follow a potential execution path $(1, 2, 4, 6, 7, 9, 10, 11)$ regardless of what input values the program is given. This is because the branching statements at lines 4 and 9 set contradicting requirements to the value of $x$.

In dynamic symbolic execution the aim is to reason about the execution paths and the inputs of a program symbolically during a concrete execution of the program. In order to execute a program $P$ symbolically, each concrete variable in $P$ is associated with a *symbolic value* in addition to its concrete value. A symbolic value represents a set of concrete values a variable can have at the current point of execution. A symbolic value of variable $x$ will be denoted by $\mathcal{S}(x)$ and it can be either:

(a) an input symbol,
(b) an expression where a binary operator is applied to two symbolic values, or
(c) an expression where a binary operator is applied to a symbolic value and a concrete value.

An *input symbol* is a symbolic representation of a single input value to a program $P$ such that no two input values have the same input symbol. The binary operators in this context mean the same ones that are used in the program with concrete variables, such as summation and multiplication. If a variable $x$ has a value that does not depend on any input values, it does not have a symbolic value associated to it. The symbolic values of variables are updated just like concrete values during execution. To be more precise, a symbolic value of a variable is constructed as follows. If the variable is assigned an input value, it will be of type (a). Copying a concrete value from a variable to another causes the symbolic value to be copied also if the variable that is being assigned to another one has a symbolic value. When a binary operator is applied to a variable that has a symbolic value, the resulting symbolic expression will be of the type (b) if the other operand has also a symbolic value and of the type (c) if the other operand is a constant or a variable that does not have a symbolic value. In the latter case, the concrete value of the constant or variable is used in the symbolic expression. It can be seen that if the input symbols in a symbolic value are replaced with concrete input values, the symbolic expression will tell what the concrete value of the variable would be at the point of execution where the symbolic value was constructed.

When a program $P$ is executed, the same execution path is followed regardless of the input values until a branching statement is encountered that selects an outgoing branch based on some variable that has a symbolic value associated with it (*i.e.*, inputs to the system affect its value). If the symbolic values of the variables that are used to determine the outgoing branch are known, it is possible to reason about the outcome symbolically. A *local constraint* is a symbolic expression $x \circ y$, where $x$ is a symbolic value, $y$ is either a symbolic or concrete value and $\circ \in \{=, \neq, <, \leq, >, \geq\}$. If it is assumed that in branching statements two values are compared (branching statements with multiple comparisons joined with logical OR or AND operators are seen as separate branching statements in this chapter), it is possible to form a local constraint for the true and false branches assuming that at least one of the variables used in the comparison has a symbolic value. A local constraint gives restrictions to the input values that must be satisfied in order for a concrete execution to take the branch corresponding to the local constraint. At any given branching statement, the two local constraints that correspond to the outgoing branches are negations of each other. Each branching statement that causes a local constraint to be constructed can be seen as a point where the set of input values following the current execution path is possibly divided into two distinct sets that follow different execution paths.

**Example 2.** Let us look at our running example program in Figure 2.1. At the beginning of any execution of the program, an input value is read to a variable $x$. Let the symbol representing this input be $input_1$. This input symbol will be the symbolic value of $x$ after the first assignment in line 1. At the next assignment statement, $x = x + 5$, the symbolic value of $x$ is updated to be $\mathcal{S}(x) = input_1 + 5$. As the following if-statement depends on $x$ and it has a symbolic value, local constraints $input_1 + 5 > 0$ and $input_1 + 5 \leq 0$ are formed to indicate what values of $x$ will follow the true and false branches of the if-statements respectively.

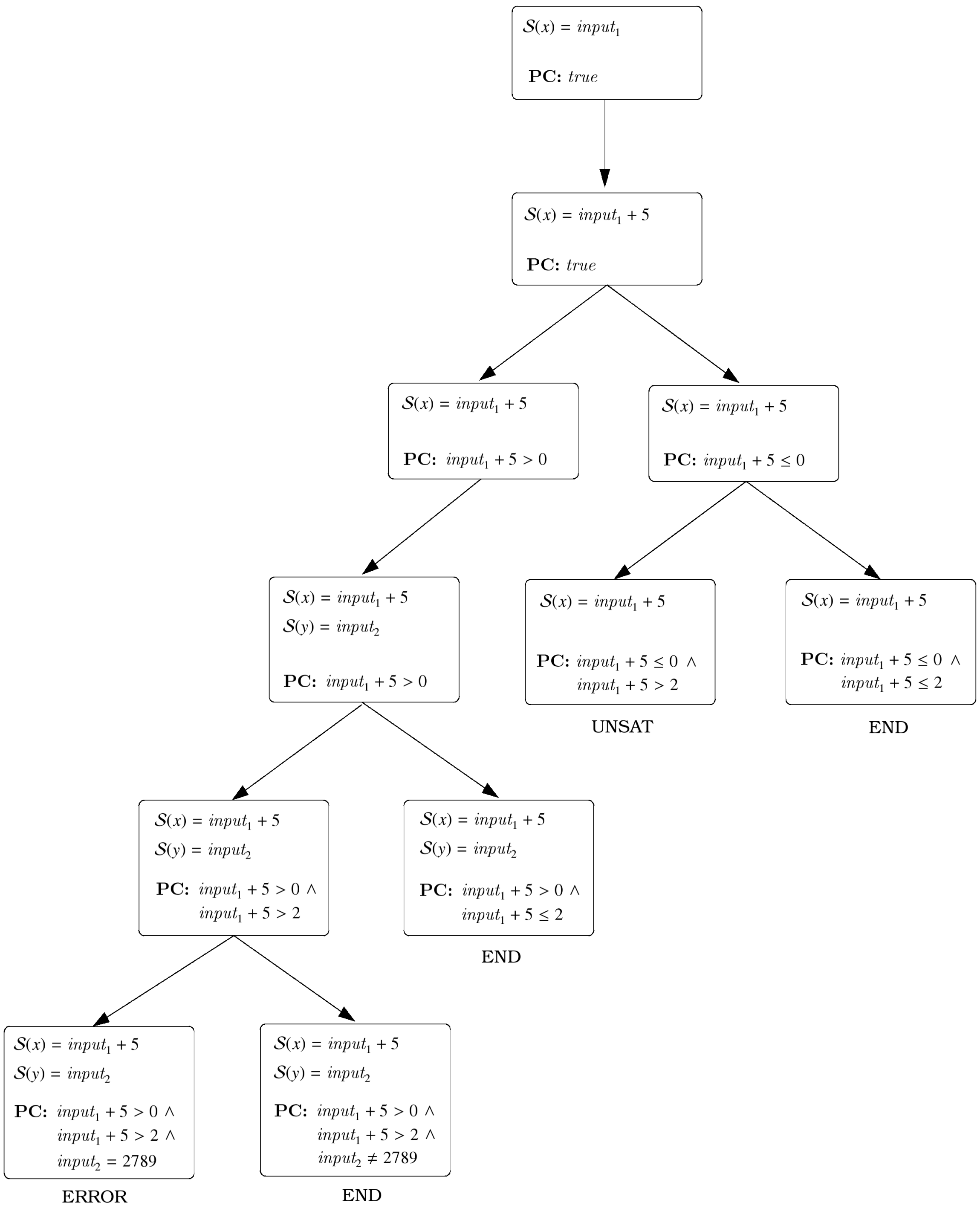
So far only the symbolic representation of primitive data types has been discussed. To be able to generate test cases for programs that take objects as input describing, for example, various data structures, the ability to reason about input references is needed as well. For primitive data types it is enough to collect arithmetic constraints as shown with local constraints but to allow reasoning about symbolic objects and their relationship to each other, it is necessary to also collect constraints that tell whether some input references must or must not point to a same object. These kind of constraints will be called *object constraints*. To make this kind of reasoning possible a symbolic value is also associated with each input object. A symbolic value of an object $o$ is denoted by $\mathcal{R}(o)$. It should be noted that if an object has a symbolic value it is always an input symbol as it is not possible to operate with object references similarly to numeric variables (*e.g.*, use summation). Object constraints are formed at branching statements where two object references are compared to each other and they are of the form $x \circ y$, where $x$ is a symbolic value of an input object, $y$ is either a symbolic value or *null* and $\circ \in \{=, \neq\}$. When input objects are constructed, they are set to reference to a same object if and only if so required by an object constraint. As input objects have data fields, they are initialised as new input values. In this work a method called *lazy initialisation* is used meaning that the fields of an object are initialised on demand only after one of the fields of a symbolic object is accessed during execution for the first time.

**Example 3.** Let us assume that a program taking two objects, $o_1$ and $o_2$, as input is executed symbolically. Let us also assume that $\mathcal{R}(o_1) = obj_1$ and $\mathcal{R}(o_2) = obj_2$. When an if-statement is executed that checks whether the references point to the same object, two object constraints are formed. $obj_1 = obj_2$ for the true branch and $obj_1 \neq obj_2$ for the false branch.

Given a prefix of an execution path, we are interested in finding concrete inputs that will exercise one of the execution paths that has the given prefix. Assuming that the local constraints and object constraints corresponding to the prefix are available, it is not enough to consider only the last constraint on the prefix as all of them can add requirements for the inputs as illustrated in the Example 1. A *path constraint* of an execution path prefix is a conjunction of all the local constraints and object constraints that must be satisfied so that the prefix can be followed by a concrete execution. If the path constraint is satisfiable, there exists concrete input values that will follow an execution path with the desired prefix and if it is unsatisfiable, then no such execution path can exist. It is, however, important to notice that the generated local constraints are constraints on unbounded integers. This means that a path constraint could be satisfiable with concrete input values that are not within the value range of the variables in the program under test. This problem could be solved, for example, by expressing the variables in local constraints as fixed-size bit-vectors but in this work the assumption is made that unbounded integers can be used.

All the possible execution paths of a program can be expressed in a form of a tree. A *symbolic execution tree* is a binary tree where the nodes represent points in an execution path where symbolic execution is occurring. An assignment with symbolic values is represented with a node that has only one child and a branching statement depending on symbolic values is represented by a node with two children, one for the true branch and the other for the false branch. The tree also contains information on the symbolic values of variables for each execution point as well as the path constraints that must be satisfied in order to reach a given node in the symbolic execution tree. The number of nodes in a symbolic execution tree can be finite or infinite depending on whether there are infinite loops in the given program.

**Example 4.** A symbolic execution tree of our example program is shown in Figure 2.2. The path constraints are denoted in the figure by the short hand PC. If the aim is to



$\mathcal{S}(x) = input_1$ PC: $true$
$\mathcal{S}(x) = input_1 + 5$ PC: $true$
$\mathcal{S}(x) = input_1 + 5$ PC: $input_1 + 5 > 0$
$\mathcal{S}(x) = input_1 + 5$ PC: $input_1 + 5 \leq 0$
$\mathcal{S}(x) = input_1 + 5$ $\mathcal{S}(y) = input_2$ PC: $input_1 + 5 > 0$
$\mathcal{S}(x) = input_1 + 5$ PC: $input_1 + 5 \leq 0 \land input_1 + 5 > 2$
UNSAT
$\mathcal{S}(x) = input_1 + 5$ PC: $input_1 + 5 \leq 0 \land input_1 + 5 \leq 2$
END
$\mathcal{S}(x) = input_1 + 5$ $\mathcal{S}(y) = input_2$ PC: $input_1 + 5 > 0 \land input_1 + 5 > 2$
$\mathcal{S}(x) = input_1 + 5$ $\mathcal{S}(y) = input_2$ PC: $input_1 + 5 > 0 \land input_1 + 5 \leq 2$
END
$\mathcal{S}(x) = input_1 + 5$ $\mathcal{S}(y) = input_2$ PC: $input_1 + 5 > 0 \land input_1 + 5 > 2 \land input_2 = 2789$
ERROR
$\mathcal{S}(x) = input_1 + 5$ $\mathcal{S}(y) = input_2$ PC: $input_1 + 5 > 0 \land input_1 + 5 > 2 \land input_2 \neq 2789$
END

**Figure 2.2:** Symbolic execution tree of the first example program

follow an execution path that takes the true branch on the first if-statement and the false branch of the second, the corresponding path condition is $input_1 + 5 > 0 \wedge input_1 + 5 \leq 2$ which is satisfiable if $input_1 = -4$ or $input_1 = -3$. Trying to follow an execution path that takes first the false branch and then the true branch in our example, leads to path constraint that is unsatisfiable. This gives the information that no execution path with the given prefix can exist. The execution path $(1, 2, 4, 6, 7, 9, 10, 11)$ discussed in Example 1 can be seen as an example of this.

## 2.2 Combining Concrete and Symbolic Execution

The purpose of this section is to give an informal description of our test generation tool that is based on the concepts introduced earlier in this chapter. As a symbolic execution tree describes the distinct execution paths of a given program, the tool is based on constructing such a tree by running the program under test both with concrete and symbolic values at the same time. If the full symbolic execution tree of a program can be constructed, it can be used to compute test inputs that give full control flow coverage of the given program.

The tool works as follows. A program under test is first modified to allow symbolic execution to be done along the concrete execution. The program is then executed first with random input values to some predefined depth. The depth limit is used in order to avoid infinite executions. During a test run, the tool keeps track of the symbolic values of variables and constructs path constrains and starts building a symbolic execution tree based on the collected constraints. Each test run can be seen as exploring one path in the symbolic execution tree of the program. As branching statements are executed, the paths in the symbolic execution tree are also extended with branches. The concrete execution follows one branch based on the concrete input values. For the other branch a new node that is marked as unvisited is added to the tree and the local or object constraint corresponding to the branch is added to the node. After a test run finishes, one of the unvisited nodes in the symbolic execution tree is selected and the path constraint corresponding to the execution path prefix the node represents is given to a constraint solver. If the path constraint is unsatisfiable, a new unvisited node is selected and if the path constraint is satisfiable, the constraint solver is asked to provide a satisfying assignment for the constraint and this corresponds to the concrete input values that are used on the next test run. When there are no unvisited branches left in the symbolic execution tree, the test generation algorithm terminates. As a result concrete input values for each of the test runs are obtained together with the information if any of these test runs caused the program to terminate due to an uncaught exception.

The described test generation approach is further illustrated by an example given below. A more formal description of how both local and object constraints are generated during execution is given in Chapter 3.

**Example 5.** Let us consider a modified version of the simple example program discussed earlier. The program now takes both a primitive integer value and an object

```
1 Class SimpleList
2 {
3     public int value;
4     public List next;
5 }
```

```
1  void example() {
2      int x = input();
3      SimpleList y;
4
5      x = x + 5;
6
7      if (x > 0)
8          y = input();
9      else
10         y = null;
11
12     if (y.next == y)
13         error();
14 }
```

**Figure 2.3:** Second example program

representing a linked list as input to the system. The example code is shown in Figure 2.3 and the different test runs from the viewpoint of a symbolic execution tree are shown in Figure 2.4. The concrete values of primitive variables are shown in parentheses as they are not part of the symbolic execution tree. The first test run is executed with randomly generated input values. The variable $x$ is assigned a concrete value $-572$ and a symbolic value $input_1$ at line 2. A node corresponding to this first symbolic operation is added as the root of the symbolic execution tree. For the $x = x + 5$ statement a new node is added to the symbolic execution tree reflecting that the symbolic value of $x$ is updated to $input_1 + 5$. At line 7 a branching statement is encountered and as $x$ has a symbolic value associated to it, two local constraints, $input_1 + 5 > 0$ and $input_1 + 5 \leq 0$, are formed for the true and false branches respectively. Two nodes are added to the symbolic execution tree to represent this branching. As the concrete execution will follow the false branch, the node corresponding to it will be chosen as the one that will be expanded by the current test run and the other node is marked to be unvisited so that future test runs can select it and compute input values that will force the execution to follow the true branch instead. Because the false branch was followed, the object reference $y$ is set to be null and this causes an null pointer exception at line 12. This causes our tool to inform the user about an found error and the current test run terminates.

After the first test run, the symbolic execution tree looks like the left most tree on Figure 2.4. The fully explored subtrees are marked with dashed lines. The resulting tree has only one unvisited node and that is selected as the node that will be expanded during the second test run. If there were multiple unvisited nodes, an arbitrary strategy could be used to select which node to expand. To get input values for the new test run the local and object constraints are collected along a path from the node to the root of the tree and a path constraint is formed of these constraints. In this case there is only one local constraint on this path and the path constraint is simply $input_1 + 5 > 0$. The path constraint is then given to an off-the-shelf constraint solver

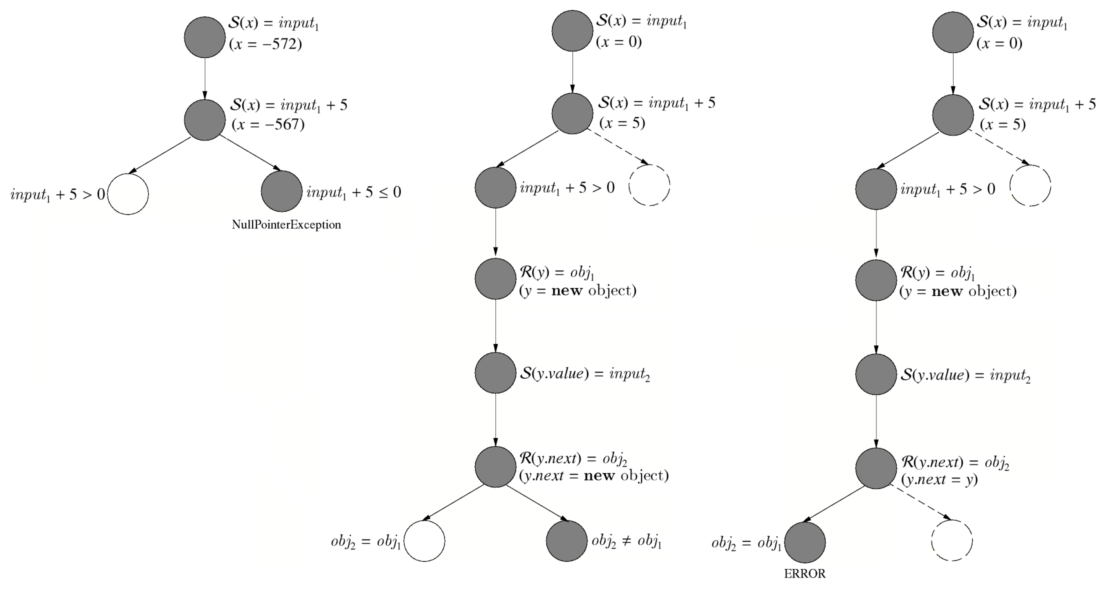

$\mathcal{S}(x) = input_1$
$(x = -572)$
$\mathcal{S}(x) = input_1 + 5$
$(x = -567)$
$input_1 + 5 > 0$
$input_1 + 5 \leq 0$
NullPointerException
$\mathcal{S}(x) = input_1$
$(x = 0)$
$\mathcal{S}(x) = input_1 + 5$
$(x = 5)$
$input_1 + 5 > 0$
$\mathcal{R}(y) = obj_1$
$(y = \textbf{new} \text{ object})$
$\mathcal{S}(y.value) = input_2$
$\mathcal{R}(y.next) = obj_2$
$(y.next = \textbf{new} \text{ object})$
$obj_2 = obj_1$
$obj_2 \neq obj_1$
$\mathcal{S}(x) = input_1$
$(x = 0)$
$\mathcal{S}(x) = input_1 + 5$
$(x = 5)$
$input_1 + 5 > 0$
$\mathcal{R}(y) = obj_1$
$(y = \textbf{new} \text{ object})$
$\mathcal{S}(y.value) = input_2$
$\mathcal{R}(y.next) = obj_2$
$(y.next = y)$
$obj_2 = obj_1$
ERROR

**Figure 2.4:** Running example of the testing algorithm

that reports that the constraint is satisfiable and gives $input_1 = 0$ as one assignment that satisfies the constraint.

Given this information the variable $x$ is given the value 0 instead of a random value on the second test run. This will cause the concrete execution to take the true branch at line 7 as expected. On line 8 an instance of the class SimpleList is read as an input. This will cause $y$ to be assigned with a new SimpleList object but the fields of this object are not initialised with input values yet. This input object is given a new symbolic value, in this case $obj_1$. When executing the line 12, the field $y.next$ is accessed and according to the lazy initialisation approach we are using this will cause all the field of $y$ to be initialised with input values. This will give the field $y.value$ symbolic value $input_2$ and $y.next$ symbolic value $obj_2$ as shown in the second tree of Figure 2.4. Whenever a new symbolic object is created and there are no restrictions placed on it in the path constraint, our tool will create a new object that is distinct from the other symbolic objects. Because of this the concrete object $y.next$ is different from $y$ and the execution follows the false branch on line 12. As the branching statement uses symbolic objects, object constraints $obj_2 = obj_1$ and $obj_2 \neq obj_1$ are created.

The second test run leaves again one unvisited node to the symbolic execution tree and the path constraint corresponding to it is $input_1 + 5 > 0 \wedge obj_2 \neq obj_1$. For the third test run the variable $x$ is given the value 0 again and when initialising the field $y.next$ it is set reference the object $y$ as the path constraint requires the objects with symbolic values $obj_1$ and $obj_2$ to be the same. With these inputs the concrete execution hits the error method call on line 13. Note that the field $y.value$ can be given a random value as it is not mentioned in the path constraint. After the third test run there are no unvisited nodes left in the symbolic execution tree as shown in the rightmost tree in Figure 2.4. This allows the test generation algorithm to terminate.

# Chapter 3

# Collecting Symbolic Constraints at Runtime

In order to give the program under test the input values computed from path constraints and to collect symbolic information about the test runs during execution, it is necessary to instrument it first. During instrumentation the original code is left unmodified so that the program can be run with concrete values and new statements are added to appropriate places to enable the symbolic execution at the same time.

In this chapter it is explained how symbolic information is stored during execution and what kind of instrumentation is needed. It is also discussed how the developed testing system can be used to test programs that take complex data (*e.g.*, objects representing data structures) as inputs and what kind of approximations the tool makes in certain cases for efficiency reasons.

The approach described in this chapter follows mostly the approach explained in [27] for tools named CUTE and jCUTE. The main differences between the approach described here and the approach taken in CUTE are highlighted at the relevant sections.

## 3.1 Syntax of Programs

As the number of different instructions available in Java bytecode and the variety of Java statements that can be written as Java source code is great enough to make the instrumentation of all the possible statement types cumbersome, it is convenient to translate Java to an intermediate language that offers less statements and has a more restricted syntax than normal Java. In our tool implementation Jimple [30] is used as this intermediate language without loss of any expressive power in comparison with Java. To describe the instrumentation process here, the full syntax of Jimple is not needed and so an idealised imperative language based on simple three-address code will be used to represent the language that is instrumented. Syntax of the statements expressible in this language is presented in Figure 3.1. In addition to the shown statements, the language also contains class and method definitions. The tools used in instrumentation are discussed in more detail in Chapter 5.

| | | |
|---|---|---|
| statement | ::= | label \| assign \| `if` comparison `goto` label \| `return` \| `begin` \| `end` |
| assign | ::= | variable = expression \| object reference = object reference |
| variable | ::= | local variable \| object field |
| object reference | ::= | object \| `null` \| `new` object \| `input` |
| expression | ::= | constant \| variable \| binop \| `invoke` \| `input` |
| binop | ::= | variable op variable |
| op | ::= | + \| - \| * \| / \| % |
| comparison | ::= | $<$ \| $\leq$ \| $>$ \| $\geq$\| == \| ! = |

**Figure 3.1:** Syntax of statements in the intermediate language

Any normal Java statement can be expressed using this simplified syntax. For example, looping constructs can be written with if and goto expressions. Note also that if-statements where logical OR operators are used (*e.g.*, `if` ($x$ == 5 ‖ $x < 0$)) must be expressed by using multiple if-statements. This has the effect that the path constraints formed during execution contain only conjunctions and no disjunctions. This is also the case with the Jimple language used in the tool. To simplify the presentation further, the types of variables and constants are left out of the discussion. The reader can imagine the variable types to be, for example, primitive Java integers but all primitive types are handled in a similar fashion as discussed in the subsequent sections.

## 3.2 Instrumentation

Adding the code for symbolic execution to a given program can be made in a fully automatic fashion with the exception that it is assumed that the user identifies the the points in the source code where the system gets its input values. If the goal is to unit test a method, the user can, for example, write a test driver that generates the wanted symbolic inputs and filters the unwanted values out if necessary and then calls the method to be tested with these values. The locations where the inputs are read in the source code are not limited in any way. This allows the user to take, for example, a fully implemented program and replace the parts where the original inputs are read with symbolic input statements and then proceed with testing.

Every statement that can read or update a variable with its value depending on the inputs must be instrumented. The approach taken in our tool is to instrument all statements that could operate on symbolic inputs regardless of whether a statement operates only with concrete values during test runs or not. This means that a majority of the lines in the code will be instrumented. This slows down the execution with a constant factor.

To describe the instrumentation process it is first discussed how symbolic values of primitive type variables and objects can be stored. The instrumentation of programs with only primitive type input values is then described. After this it is explained how the instrumentation can be extended for programs that take objects as inputs and finally it is discussed how method calls must be instrumented.

### 3.2.1 Storing Symbolic Values

To form the symbolic path constraints introduced in the previous chapter, it is necessary to know for each variable the symbolic expression associated with it during execution. For this reason we introduce symbolic memory maps $\mathcal{S}$ and $\mathcal{R}$ for primitive type variables and objects respectively. These memory maps are maintained by the code added during instrumentation. A symbolic value of a variable $x$ is denoted by $\mathcal{S}(x)$ like in the previous chapter and $\mathcal{S}' = \mathcal{S}[x \mapsto s]$ is written to express that $\mathcal{S}'(x) = s$ and the the rest of the mappings in $\mathcal{S}$ and $\mathcal{S}'$ are identical. $\mathcal{S}' = \mathcal{S} - x$ is used to denote that the mapping of variable $x$ to a symbolic value is removed. $\mathcal{S}'(x)$ is defined in this case to return an implementation specific value indicating that $x$ is not symbolic. The map $\mathcal{R}$ maps objects to symbolic values in a similar fashion. In addition to $\mathcal{S}$ and $\mathcal{R}$ a mapping denoted by $\mathcal{M}$ is also maintained but the description of this mapping is postponed until Section 3.2.3

On the implementation level the memory addresses of data values could be used as the keys to which the symbolic values are mapped to as the addresses can be seen as unique identifiers. However, because in Java it is not possible to have access to pointers and memory addresses, the names of the variables are used as the keys for primitive type variables. This solution has naturally the problem that names are not unique for each variable. Therefore the mapping has been implemented by adding a new symbolic variable during instrumentation for each primitive local variable or object field in the program. In other words, each primitive type variable has a counterpart variable with the same exact scope as the original. As in each scope we have no ambiguity over the memory address a variable name refers to, the problem of non-unique names is solved. This approach naturally requires that the mapping is maintained by the new variables added during instrumentation and this effectively doubles the number of variables the program uses. By applying static analysis to the source code to identify the statements and variables that can be affected by the inputs (*e.g.*, using type-dependence analysis [2]), it would be possible to optimise the amount of instrumentation to gain some improvement in execution time and memory usage. It should be noted here that the described method to store symbolic values associates the symbolic value with a variable and not with the value. As in Java it is not possible to have two primitive type variables to point to the same memory location, it is safe to do the association this way (*i.e.*, it is not possible to have aliasing variables changing their values without changing the symbolic value of the counterpart).

With objects, on the other hand, it is possible to have multiple references to the same object. However, with objects it is also possible to use a reference as an identifier. Therefore the mapping $\mathcal{R}$ is implemented by maintaining a data structure that maps an object reference to a symbolic value. Whenever a symbolic value of an object is needed, the data structure is searched for a reference to the object and the symbolic value mapped to the reference is returned if the reference is found.

To keep the symbolic values stored for each variable short during execution, each time a symbolic value is changed a new symbolic identifier is made to represent this value. For example, instead of storing a symbolic value $input_1 + 5$ for a variable $x$ a symbolic value $s_0$ is used and the information that $s_0 = input_1 + 5$ is maintained separately. After another summation with value 5 the symbolic value would be $\mathcal{S}(x) = s_1$ and $s_1 = s_0 + 5$. Consider a case where a variable is summed repeatedly with itself. This would lead to the symbolic expression growing exponentially in the number of summations if new symbolic identifiers were not introduced. Naturally it is possible to do some simplifications like $input_1 + input_1 = 2 \times input_1$ similarly to [27, 29] to keep the symbolic expressions succinct. This possibility is discussed further in Chapter 4 but the implementation of such simplifications is left for future work.

The symbolic execution tree constructed during test runs is maintained in a separate module to the runtime environment where the tests are executed. Our testing system can be seen as consisting of two parts: one is the instrumented program under test and the second is a module that maintains a symbolic execution tree and uses it to select test inputs for test runs. An instrumented program will be called a *test executor* and the second module a *test input selector*. The test executors store data relevant to a single test run by themselves and report all the information relevant to the construction of the symbolic execution tree to the test input selector. The details of the test input selector are given in Chapter 4.

### 3.2.2 Symbolic Execution with Primitive Data Types

To instrument a given program, each statement in it will be processed one at a time and code performing symbolic execution will be added for that statement if needed. The basic principle during instrumentation is to try to minimise the amount of code added directly to the original code and to do most of the work in methods that are called from the instrumented code lines. The statements added during instrumentation to a program containing only primitive type input values are shown in Table 3.1. The letter $v$ will be used as a shorthand for variables, letter $o$ for objects and letter $e$ for expressions.

Before a test run can be started, the symbolic execution part of the program must be initialised. During the initialisation, a connection is formed to the test input selector. The selector sends a list of concrete input values, that must be used one at a time in the given order when an input statement is executed to make sure that the correct execution path prefix will be followed. The input values received from the test input selector are denoted by a mapping $\mathcal{I}$ that maps the number of an input (expressed by *inputNumber*) to a concrete input value. In other words, the map $\mathcal{I}$ can be seen as a sequence of input values ordered by the input number (*i.e.*, when the first input statement is executed the first value of the sequence is used). If the program is not fully deterministic (*e.g.*, some of the inputs are not replaced with symbolic input statements) the test run is not guaranteed to follow the expected execution path. To

| Before instrumentation | After instrumentation |
|---|---|
| `begin` | $\mathcal{I}$ = receive inputs;<br>$i = k = j = inputNumber = 0$;<br>$\mathcal{S} = \mathcal{M} = \mathcal{R} = []$;<br>`begin` |
| $v = e$; | $\textsc{executeAssignment}(v, e)$;<br>$v = e$; |
| $v$ = `input;` | $v = \textsc{getInput}(v)$; |
| `if` $v_1$ $op$ $v_2$ `goto` $l$; | $\textsc{executeCondition}(op, v_1, v_2)$;<br>`if` $v_1$ $op$ $v_2$ `goto` $l$; |
| `goto;` | $\textsc{checkGotoCount}()$;<br>`goto;` |
| `end` | $\textsc{reportEnd}()$;<br>`end` |

**Table 3.1:** Instrumentation of statements

detect executions that do not follow expected execution paths, a bit-vector containing the information of which outgoing branch was taken at each branching statement is constructed. By reporting this bit-vector to the test input selector it can be checked if the correct path has been followed. This is further discussed in Chapter 4.

At every point where the program terminates successfully, the test selector needs to be informed about this so that an execution path can be marked to be explored. Every time a program terminates, normally or due to an error, the connection to the selector is closed. If the successful termination is not reported before this happens, the test run is considered to have found a program error. This means that every non-error termination point must be identified, which in our tool is done automatically when the program is transformed into Jimple.

Every input statement indicated by the user is replaced with a call to a method that assigns a concrete input value and a symbolic value to the respective variable. The symbolic variable is simply assigned with a new unique input symbol and it is reported to the test input selector that this value represents the new input value. For the concrete input, it is first checked if there are values given by the selector left to use. If there are not, a random value will be used. This is further illustrated in Figure 3.2.

With every assignment statement it is necessary to make sure that the symbolic values are also updated. Figure 3.3 shows how this is done. When a concrete value is assigned to a variable, the symbolic value associated with the variable is simply removed if such a value exists. Assigning a value from a variable to another requires only copying the symbolic value to the respective symbolic variable. The case $v = v_1\ op\ v_2$ where the result of applying a binary operator to two variables is assigned to another variable is slightly more complex. It is first checked (on line 10) whether the binary operator is supported by the constraint solver being used (*e.g.*, our tool does not support symbolic execution of the operator %) and if it is not, the assignment is executed only concretely

```
getInput($v$)
 1  $\mathcal{S}[v \mapsto s_i]$; // $s_i$ is a new symbolic value
 2  $i = i + 1$;
 3  report($\mathcal{S}(v) = input_k$);
 4  $k = k + 1$;
 5  if ($inputNumber \in$ domain($\mathcal{I}$))
 6      $result = \mathcal{I}(inputNumber)$;
 7  else
 8      $result$ = a random value;
 9  $inputNumber = inputNumber + 1$;
10  return $result$;
```

**Figure 3.2:** Getting correct input values during execution

and the symbolic value of $v$ is removed. If the binary operator is applied to at least one variable with a symbolic value, a new unique symbolic identifier is generated to express the result of the assignment and this identifier is given as the symbolic value of $v$. As the symbolic identifier in itself does not contain the information of the symbolic expression corresponding to it, this fact is reported to the test input selector as shown, for example, on line 18. If the binary operator is a multiplication or division and the both variables to which the binary operator is applied to have symbolic values, the constraints resulting from using the new symbolic value are nonlinear. As the nonlinear integer programming problems are in general undecidable, there might be no support in constraint solvers to handle nonlinear constraints that result from these kind of assignments. (However, when representing variables as fixed-size bit-vectors, which corresponds more closely to the way values are stored in computers, the problem becomes decidable.) If the tool is used with a constraint solver that does not support nonlinear constraints, the symbolic expression corresponding to the assignment is approximated by replacing one of the symbolic values used in the assignment with a concrete value (line 16). This allows part of the symbolic information to be carried over the assignment statement. This same approximation is also used in the jCUTE tool. In its current state our tool does not yet support the use of bit-vectors and all values are considered to be unbounded integers.

To execute if-statements symbolically, it is first determined which outgoing branch the concrete execution will take and then a local constraint is reported to the test input selector as illustrated in Figure 3.4. The method first checks whether the if-statement operates on symbolic values or not. If only concrete values are used, the test input selector needs not to be informed as the statement does not affect the symbolic execution tree of the program. If symbolic values are used, the local constraint for the true branch is reported to the test input selector (the false branch can be obtained by simply negating this condition) as well as the branch taken by the current concrete execution so that the test input selector knows which of the branches it will keep expanding. Note that on line 2 the information of the taken branch is used in con-

```
EXECUTEASSIGNMENT($v, e$)
match ($e$)
    case $c$: //$c$ is a constant value
        $\mathcal{S} = \mathcal{S} - v$;
    case $v_1$: //$v_1$ is a variable
        if ($v_1 \in$ domain($\mathcal{S}$))
            $\mathcal{S} = \mathcal{S}[v \mapsto \mathcal{S}(v_1)]$;
        else
            $\mathcal{S} = \mathcal{S} - v$;
    case $v_1$ op $v_2$:
        if (op $\notin \{+, -, *, /\}$)
            $\mathcal{S} = \mathcal{S} - v$; //operators like % are unsupported
        else if($v_1 \in$ domain($\mathcal{S}$) $\wedge$ $v_2 \in$ domain($\mathcal{S}$))
            $\mathcal{S}[v \mapsto s_i]$; //$s_i$ is a new symbolic value
            $i = i + 1$;
            if (op $\in \{*, /\}$ $\wedge$ constraint solver supports only linear constraints)
                REPORT($\mathcal{S}(v) = \mathcal{S}(v_1)$ op $v_2$);
            else
                REPORT($\mathcal{S}(v) = \mathcal{S}(v_1)$ op $\mathcal{S}(v_2)$);
        else if($v_1 \in$ domain($\mathcal{S}$))
            $\mathcal{S}[v \mapsto s_i]$;
            $i = i + 1$;
            REPORT($\mathcal{S}(v) = \mathcal{S}(v_1)$ op $v_2$);
        else if($v_2 \in$ domain($\mathcal{S}$))
            $\mathcal{S}[v \mapsto s_i]$;
            $i = i + 1$;
            REPORT($\mathcal{S}(v) = v_1$ op $\mathcal{S}(v_2)$);
        else
            $\mathcal{S} = \mathcal{S} - v$;
```

**Figure 3.3:** Executing symbolic assignments

```
EXECUTECONDITION($op, v_1, v_2$)
1  $branchTaken$ = EVALUATE($v_1$ $op$ $v_2$);
2  CONSTRUCTBRANCHBITVECTOR($branchTaken$);
3  if ($v_1 \in$ domain($\mathcal{S}$) $\wedge$ $v_2 \in$ domain($\mathcal{S}$))
4      REPORT($\mathcal{S}(v_1)$ $op$ $\mathcal{S}(v_2)$, $branchTaken$);
5  else if($v_1 \in$ domain($\mathcal{S}$))
6      REPORT($\mathcal{S}(v_1)$ $op$ $v_2$, $branchTaken$);
7  else if($v_2 \in$ domain($\mathcal{S}$))
8      REPORT($\mathcal{S}(v_2)$ $op$ $v_1$, $branchTaken$);
```

**Figure 3.4:** Executing if-statements symbolically

struction of the bit-vector containing all the chosen branches regardles whether the if-statement operates on symbolic values or not. The constructed bit-vector is sent to the test input selector whenever REPORT method is called.

As a program may have infinite execution paths due to looping constructs, the test runs must be cut at some predefined depth to make sure that the testing terminates. The only ways of creating a loop in Jimple and in our idealised language is by using `goto` statements or recursive method calls. Each `goto` statement is instrumented with a CHECKGOTOCOUNT method, that implements a counter that is increased each time a `goto` statement is executed. Every method call is instrumented with a CHECKINVOKECOUNT method, that works similarly to the case with `goto` statements. Whenever a method or goto counter exceeds a given depth value, the test run is reported to have been successful and the test run is terminated.

### 3.2.3 Symbolic Execution with Objects

In this section the instrumentation process is extended for programs that use objects as inputs to the system under test. Our tool supports two different ways of how input objects are created and how object constraints resulting from the use of these objects are constructed. The first way collects object constraints during execution and is described next. The second way is a simplified version that does not generate object constraints automatically but also does not require as much instrumentation.

#### Symbolic Execution with Object Constraints

The additional instrumentation needed for symbolic execution with input objects is shown in Table 3.2. Notice that there is no need to add any instrumentation to assignment statements using object references. To see the difference to the primitive data type case, consider assignments $x = 5$ and $y = null$, where $x$ is an integer and $y$ is an object reference. The first assignment replaces the earlier value in the memory location reserved for variable $x$. If a symbolic value is associated with the memory location, assignments affecting the value in it must also affect the symbolic value. In the second assignment the value of $y$ is not an object but a reference to one. This means

| **Before instrumentation** | **After instrumentation** |
|---|---|
| $o$ = `input`; | $o$ = GETSYMBOLICOBJECT($o$); |
| $v$ = $o.field$; | LAZYINITIALIZE$(o)$;<br>EXECUTEASSIGNMENT($v, o.field$);<br>$v$ = $o.field$; |
| $o.field$ = $e$; | LAZYINITIALIZE$(o)$;<br>EXECUTEASSIGNMENT($o.field, e$);<br>$o.field$ = $e$; |
| *if* $o_1$ *op* $o_2$ *goto* $l$; | EXECUTEOBJECTCONDITION(*"op"*, $o_1, o_2$);<br>*if* $o_1$ *op* $o_2$ *goto* $l$; |

**Table 3.2:** Instrumentation of object statements

that when $y$ is set to be *null*, it does not change the object it was referencing in any way. As symbolic values are associated with objects, no object reference assignments can change the symbolic values of objects. Furthermore it is not possible to directly replace or delete an object in a given memory location in Java.

Getting objects as inputs is more complicated than getting simple numeric values. The main difference to primitive type inputs is that with primitive inputs the test input selector can simply give a concrete value that is assigned to a variable but with object inputs no such concrete value can be given. In place of concrete values the test input selector sends *logical addresses* to input objects. A logical address is a natural number where the value zero is a special value that corresponds to a null reference. When the test input selector wants to have two input objects to be the same, it will give them both the same logical address in the input map $\mathcal{I}$. For example, if the input map corresponds to an input sequence (1,0,1,2), the first and third calls to GETSYMBOLICOBJECT will give the reference to the same object, second call will give a null reference and fourth call a reference to an object that is not the same as the ones given by the earlier calls. To be able to return a reference to an already created object, as in case of the third call in the previous example, a mapping from logical address to concrete objects is maintained. This mapping is denoted by $\mathcal{M}$.

Figure 3.5 shows the algorithm for creating a new input object. Similarly to the primitive input case it is first checked if the test input selector has given an input value that must be used at the current execution point. If the input map contains a value, it corresponds to a logical address of an object. If the address corresponds to a null reference, the algorithm simply returns null as the result. Otherwise it is checked (line 5) if the required object has already been created by looking a reference to it in the map $\mathcal{M}$. If the reference is found, it is returned as the result and if it is not found, a new object is created and the mapping from the given logical address to the newly created object is added to $\mathcal{M}$. A symbolic value $obj_j$ is also associated to the created object. The value $j$ is a running number to prevent the same input symbol from being used multiple times.

If the input map does not contain a logical address to be used, a new object is created and a symbolic value is associated with it. Notice that in this case no logical address

```
getSymbolicObject($o$)
 1  if ($inputNumber \in$ domain($\mathcal{I}$))
 2      $l = \mathcal{I}(inputNumber)$;
 3      if ($l == 0$)
 4          $result$ = null;
 5      else if($l \in$ domain($\mathcal{M}$))
 6          $result = \mathcal{M}(l)$;
 7      else
 8          $result$ = new object of type $o$;
 9          $\mathcal{R} = \mathcal{R}[result \mapsto obj_j]$;
10          $\mathcal{M} = \mathcal{M}[l \mapsto result]$;
11  else
12      $result$ = new object of type $o$;
13      $\mathcal{R} = \mathcal{R}[result \mapsto obj_j]$;
14  $inputNumber = inputNumber + 1$;
15  $j = j + 1$;
16  return $result$;
```

**Figure 3.5:** Getting symbolic objects as inputs

is given to the object. This is because all input objects are assumed to be distinct unless required otherwise by the test input selector. As the input map does not containt at this point any new values, there cannot be any requirements for input object created later during the test run. Note also that any new object returned by getSymbolicObject is simply created by using a default class constructor. This means that the fields of the object are not initialised with any symbolic values at this point.

In lazyInitialize, the given object is marked to have been initialised to prevent multiple initialisations. To initialise the fields, the tool supports two approaches: all of the fields in an object can be initialised as new symbolic inputs or the user can provide a custom method that has been added to the class of the object and does the initialisation. The first approach creates objects that have no restrictions on what values their primitive type fields can have and that have every object field set to be a new symbolic object. This approach is suitable for simple objects that do not have dependencies between their fields, such as linked lists that are not sorted with respect to their content. However, when the objects are more structured and have invariants that must hold, it is more convenient to allow the user to specify which fields are to be initialised with symbolic inputs and allow some fields to be initialised with only concrete values, possibly depending on the input values received for the other fields. This is achieved by calling a user written method at the time when lazy initialisation is done. Currently the method must be added manually to the source code of the class but this could be done automatically during instrumentation even when no source code of the object class is available. The initialisation method can

```
EXECUTEOBJECTCONDITION($op, o_1, o_2$)
1  if ($op \in \{==, !=\}$)
2      $branchTaken$ = EVALUATE($o_1$ $op$ $o_2$);
3      CONSTRUCTBRANCHBITVECTOR($branchTaken$);
4      if ($o_1 \in$ domain($\mathcal{R}$) $\wedge$ $o_2 \in$ domain($\mathcal{R}$))
5          REPORT($\mathcal{R}(o_1)$ $op$ $\mathcal{R}(o_2)$, $branchTaken$);
6      else if($o_1 \in$ domain($\mathcal{R}$) $\wedge$ $o_2$ == null)
7          REPORT($\mathcal{R}(o_1)$ $op$ null, $branchTaken$);
8      else if($o_2 \in$ domain($\mathcal{R}$) $\wedge$ $o_1$ == null)
9          REPORT($\mathcal{R}(o_2)$ $op$ null, $branchTaken$);
```

**Figure 3.6:** Generating object constraints

access normally the public and private fields of the object and may contain arbitrary Java code. The user is, however, responsible that the initialisation code does not have side effects that could not happen when the original program is executed without any code added for symbolic execution. The problem of creating input objects that must satisfy invariants is discussed in more detain in the Section 3.4. The lazy initialisation approach is one of the biggest differences in our tool in comparison with jCUTE. In jCUTE all input objects are initialised as null references on the first time they are encountered in a test run and they are initialised with randon inputs like in our tool if a local constraint requires them to be non-null. The initialisation is done at the point where the object is received as input and not on demand as in our tool. The advantages and disadvantages of lazy initialisation in comparison to the jCUTE method are discussed in Chapter 4.

To collect object constraints the EXECUTEOBJECTCONDITION method presented in Figure 3.6 is executed before any if-statement that compares objects references instead of primitive values. Our tool collects object constraints that can be only of the form $o_1 = o_2$, $o_1 \neq o_2$, $o_1 = null$ and $o_1 \neq null$, where $o_1$ and $o_2$ are symbolic objects. The EXECUTEOBJECTCONDITION method checks if the comparison of objects falls under one of these types and also determines the outgoing branch of the if-statement taken by the concrete execution (in line 2). If the comparison is of the supported type and input objects are used in the comparison, the method generates an object constraint based on the symbolic values of the objects. As null object references do not have symbolic values associated to them, they are handled as a special case (in lines 6 and 8). The generated object constraint and the branch that the concrete test run will take are then reported to the test input selector.

### Symbolic Execution without Object Constraints

In the simplified mode of our tool no object constraints are collected during symbolic execution. In this mode all the symbolic objects returned by GETSYMBOLICOBJECT method are unique non-null objects and the fields of the symbolic objects are initialised lazily as normal. In this mode there is no need to add EXECUTEOBJECTCONDITION methods during instrumentation and path constraints are simplified as the object constraints are missing completely. This approach is suitable in cases where the user wants to write his own test driver that generates the dependencies between input objects and ignore the dependencies that the tool can detect automatically during testing. For example, if a user is interested of testing a program only with input linked lists that have no cycles, the simplified mode can be used as the symbolic execution ignores the possible execution paths that could be explored with cyclic lists. Also if the program under test receives only primitive data type inputs, this mode performs slightly faster as there is less instrumentation involved.

### 3.2.4 Symbolic Execution with Method Calls

In Java all arguments to methods are passed by value. This means that when a method is called with arguments that have symbolic values associated with them, the symbolic values must be associated with the corresponding new variables inside the method as well. Table 3.3 shows the code instrumented at method calls. When a method is called, all symbolic values of the arguments are pushed into an argument stack and these values are read from the stack and assigned to the corresponding symbolic variables at the beginning of the method execution. Likewise, the symbolic return value of a method is transferred from the method to the caller using the argument stack. To be more precise, the tool must also be able to handle cases where the method caller and the method might not be both instrumented as the user has control of what parts are instrumented and there might be some libraries or native methods that cannot be instrumented. Otherwise the stack could be empty when it is read or there might be some old argument values that were not read and removed by an uninstrumented method. For this the tool associates a method identification to the elements in the argument stack and uses this to check that there are no old arguments and removes them if necessary when adding new ones and makes sure that the arguments or return values received are in fact from the right source. If during a pop operation the argument stack is empty or contains old arguments, no symbolic values are passed to a method or a caller. The basic principle behind method instrumentation, however, stays the same as shown in Table 3.3 regardless of the implementation of these checks.

Note also that when object references are used as arguments in method calls or as return values there is no need for additional instrumentation. The same reasoning as with assignment statements holds in this case as well.

## 3.3 Used Approximations

The path constraints generated by running a program that has been instrumented the way it has been described in this chapter are not guaranteed to be precise enough to make it possible to generate test inputs that will cover all the possible behaviour of the program. That is, the current approach cannot be used in general for proving that an implementation does not throw any uncaught exceptions. This limitation is due to the fact that local or object constraints are only constructed at branching points of a program similarly to [27] where a program code, a[i] = 0; a[j] = 1; if (a[i] == 0)

| **Before instrumentation** | **After instrumentation** |
|---|---|
| $v = method(v_1, ..., v_n);$ | CHECKINVOKECOUNT();<br>$push(\mathcal{S}(v_1)); ...; push(\mathcal{S}(v_n));$<br>$v = method(v_1, ..., v_n);$<br>$\mathcal{S}[v \mapsto pop()];$ |
| $method(v_1, ..., v_n)$ {<br>...<br>**return** $v$;<br>} | $method(v_1, ..., v_n)$ {<br>$\mathcal{S}[v_n \mapsto pop()]; ...; \mathcal{S}[v_1 \mapsto pop()];$<br>...<br>$push(\mathcal{S}(v));$<br>**return** $v$;<br>} |

**Table 3.3:** Instrumentation of methods and method calls

ERROR, was given as an example of this. If the variables $i$ and $j$ are input variables, it is possible to follow execution paths that either hit the error statement or avoid it based on the fact whether $i$ and $j$ are equal. However, as there is no if-statement that would check for this fact, our tool assumes that the values are independent and does not generate constraints $i = j$ and $i \neq j$. Similarly, our tool does not generate null dereferences or object reference aliases if there are no branching statements that result in such object constraints. This no aliasing assumption is made also by CUTE and jCUTE to improve efficiency as in many practical cases the approximation seems to give reasonably good results. EXE [8], however, is a symbolic execution based tool that creates exact constraints even when aliasing as shown above can occur. The approach used is to add a disjunction of all possible aliasing cases to the path constraint. The downside is that the path constraints will get more complex and thus more difficult for the constraint solver to solve. Improving the accuracy of the constraints generated by our tool is left for future study.

Also the tool might not notice that when a value is divided by a variable having a symbolic value, it is possible that the set of concrete values the symbolic value represents might containt the value zero that will cause division by zero errors. This can be easily corrected by asking the constraint solver to check whether the symbolic value can be zero, given the current path constraint and reporting an error if this is the case. However, this could be expensive if the program contains a large number of divisions and so we have left the support these checks for future work.

Another point where the tool fails to explore all possible behaviour is when symbolic values are used with operators that are not supported by the used constraint solver or uninstrumented methods are called, such as native calls to functions implemented in another programming language. As discussed earlier, the symbolic values are approximated with concrete values in these cases. Such approximations are necessary and actually show the advantage that dynamic test generation has over static methods. To illustrate this, consider that we are unit testing the following method:

```
boolean test (int x, int y) {
    if (x == blackBox(y))
        return true;
    else
        return false;
}
```

The method *blackBox* is an uninstrumented method and no source code for this method is available. If static analysis is used to this method, it is impossible to generate concrete input values for $x$ and $y$ that will test either of the branches in the *test* method as nothing can be assumed about the value returned by *blackBox*. Our tool can, similarly to other dynamic symbolic execution tools, circumvent this problem partly. As the program is first executed with concrete random values, some concrete value is also received as the result of *blackBox(y)* and a local constraint that forces $x$ to be equal or unequal to this value can be generated. Therefore it is possible to find input values for both of the branches. Naturally, the symbolic value associated with $y$ is lost and our tool is limited to enter the branches with only some random concrete values.

The use of unbounded integer values in local constraints is also an approximation. The tool can fail to generate correct input values for an execution path where a concrete value associated with a symbolic value overflows or underflows. For example, consider that a variable the has maximum concrete value and a symbolic value. If the value of the variable is increased by one, the value overflows but this is not seen in the symbolic value in any way. If for the next test run the test input selector gives a slightly different input value such that the variable in question has a smaller value, the overflow does not happen and the test run could follow an unpredicted execution path.

## 3.4 Objects with Invariants

Generating object inputs that must satisfy any invariant causes problems if the fields of an object can be initialised with arbitrary input values. For example, consider a case where a method that gets a binary search tree as input is unit tested. In binary search trees, at any given node in the tree, the left subtree of the node contains only values that are less than the value of this node and the right subtree contains values that are greater or equal to the value of the node. As there are no restrictions on how the concrete input values are generated other than the path constraints collected during testing, it could happen that a binary search tree that does not fulfil the invariant that requires the nodes to be correctly ordered, is generated. If the method being tested is assuming that the input it receives is in fact a valid binary search tree, the testing could generate test cases that are not possible with valid inputs. This could lead to the tool to report unwanted errors.

Using the initialisation method approach described in the previous section can help in some cases. For example, if an input object has two fields of which the first has to be always greater than the second, it is easy write an initialisation method that adds a requirement to the path constraint that guarantees that the fields are initialised correctly. However, this approach might not be enough if the object that is being initialised is part of a data structure consisting of many objects. The initialisation method can look at the data structure only from the point of view of the object itself, but to create a valid data structure as an input, it might be necessary to look at the data structure as a whole. For example, in the binary search tree, if the node object does not contain a reference to its parent node, it is impossible to constrain the value of the node to be less or greater than the value of the parent as the initialisation method does not have access to the parent. To generate such data structures, the user must write an external test driver that generates valid inputs first and only after that passes the structure to the method that is being tested. The price to pay here is that the input structure will be initialised at least partly before it is used regardless of the requirements that the system under test might place on the structure. In the initialisation method approach the object will be initialised on demand and this can avoid generating some unnecessary test cases. For example, with an external test driver it might be necessary generate binary search trees of all possible shapes to guarantee that the testing is exhaustive even if some shapes would follow the same execution path.

In [31] two approaches are presented for generating input data structures. Both of these approaches can be used with our tool by writing a test driver that implements the approach. In the first approach, the data structures are generated by using repeatedly the basic methods the data structure offers (*e.g.*, creation of a new data structure and addition of a new element) if these are available. A data structure constructed this way with symbolic elements is valid if the implementation of it is correct. It is important to note here, that to generate all possible variants of a data structure, it may be necessary to use all the operations the data structure provides. For example, it is possible that some variants cannot be generated by using only additions but element removals are also necessary.

The second approach is to use a method that checks if an required invariant holds in a given data structure. When this method is executed with a symbolic input, the local and object constraints added to the path constraint during the checking ensure that the objects passing the check are representing valid data structures. In a way, this approach can be seen as solving the method for checking invariants: the symbolic execution of the method generates the constraints that lead to valid structures and the execution paths leading to invalid structures can be terminated before the generated structure is passed to the system under test.

# Chapter 4

# Generating Inputs

In this chapter the focus is moved to describing the second main part of our tool, the test input selector, which receives the constraints generated during test runs and uses these constraints to compute new input values. Many of the previously implemented dynamic symbolic execution tools, such as [27, 17], use information only from the latest test run. This, in practice, limits the order in which a symbolic execution tree of a program is explored to a depth first search but on the positive side saves memory as there is no need to construct a representation of the symbolic execution tree. Our tool, however, constructs and maintains a symbolic execution tree based on the information of all the previous test runs. A similar approach is also used in a more recent tool called Pex [29]. Constructing a symbolic execution tree allows the test input selector to use a variety of different strategies on how to choose an unvisited branch from the tree for the next test run. It also makes it possible to run multiple test runs concurrently as the test runs exercise different execution paths and the test runs need only communicate with the test input selector and not with each other. This will be further discussed in this chapter and in Chapter 5.

The general working principle of the test input selector is shown in Figure 4.1. The test input selector uses a strategy $S$ to select an unvisited node from the symbolic execution tree. The search strategies implemented in our tool are discussed in Section 4.2. After an unvisited node is selected, a path constraint corresponding to it is formed by collecting local and object constraints on the path from the unvisited node to the root of the tree (lines 6-8). For the first test run the path constraint is considered to be true so that the first test run will be executed with unconstrained input values. To get concrete input values for a desired execution path the path constraint is given to an off-the-shelf constraint solver. Solving the path constraints is discussed in more detail in Section 4.3. In particular, it will be discussed how object constraints are solved and what implications they might have on the symbolic execution tree, as the lazy initialisation approach used by the test executors can cause some complications if these are not taken into account. If the path constraint is satisfiable, the concrete input values that satisfy the constraint are given to a test executor and the symbolic execution tree is updated based on the messages generated during the test run.

How the symbolic execution trees are represented and updated based on the received messages are discussed in the following section. Because of the way how our tool

```
1   Tree $T$ = new symbolic execution tree;
2   Strategy $S$ = select search strategy;
3   while ($T$ has unvisited nodes)
4       $m = n = S(T)$; //an unvisited node $n$ is selected by using strategy $S$
5       $pc = \top$;
6       while ($m \neq T.root$)
7           $pc = pc \wedge m.constraint$; //a path constraint $pc$ is constructed
8           $m = m.parent$;
9       $inputs$ = SOLVE($pc$);
10      if ($inputs \neq$ unsatisfiable)
11          Give $inputs$ to a test executor $e$;
12          Expand $n$ based on symbolic execution done by $e$;
13          if ($e$ reports an error)
14              Report input values leading to error;
15      else
16          CHECKALTERNATIVE($n$);
17      mark $n$ visited;
```

**Figure 4.1:** General testing algorithm

constructs symbolic execution trees, it is sometimes necessary to modify the tree when a path constraint is found to be unsatisfiable as hinted by line 16 in Figure 4.1. The purpose of this method call will also be explained in the following section.

## 4.1 Representing Symbolic Execution Trees

The data structure for the symbolic execution tree can vary from a search strategy to another as each strategy might need to have some information stored to the tree that no other strategy requires. Any data structure that is used to maintain the information needed to compute new input values must have the following three characteristics:

- it can be used to get path constraints for currently unvisited branches,
- it can be used to check that no previously visited branches will be added to the set of unvisited branches, and
- it can be used to check that a test run follows the execution path predicted by the branches in the symbolic execution tree.

The first characteristic is self explanatory, the main purpose of a symbolic execution tree is to provide the path constraints so that concrete input values can be computed. The second characteristic is necessary so that it can be quaranteed that no symbolic execution path is explored multiple times. Finally, the third characteristic is needed

| Message type | Description |
|---|---|
| *Assignment* | A new symbolic value has been created due to a new input or assignment statement that uses a symbolic value with a binary operator. |
| *Branch* | Symbolic value has been used at a branching statement. Contains a local or object constraint and the branch that was taken by the concrete execution. |
| *Object Initialisation* | A symbolic object has been initialised. |
| *Goto limit* | The number of goto statements executed during a test run has exceeded the given limit and the test run has been terminated as a result. |
| *Method call limit* | The number of method calls executed during a test run has exceeded the given limit. |
| *Error* | The Java program under test has terminated due to an uncaught exception. |
| *End* | The program has terminated normally. |

**Table 4.1:** Message types and their descriptions

because it is possible due to approximations our tool makes that a test run does not follow the execution path that would be expected after solving a path constraint of an unexplored branch in the symbolic execution tree. Handling the executions that fail to follow the predicted path is discussed in more detail later in this chapter.

In principle it is possible to simply store path constraints of the unvisited branches, for example, to a list instead of a tree and select one of them when a new test run requires input values. However, this would require that in each element in the list there is enough information in addition to the path constraint so that it is possible to check the two latter characteristics listed above. By using trees it is not necessary to duplicate any data of a given node to any successor nodes in its subtree. This allows us to save some memory. All the strategies currently implemented in our tool use binary trees as the main data structure and even though there are some differences in the fields that each node in the tree has, the basic principle how the trees are constructed is the same.

It will be described next how a node in a symbolic execution tree is expanded based on the messages received from test executors and after that the problem of executions failing to follow the correct path is looked into.

## 4.1.1 Constructing Symbolic Execution Trees

As illustrated by the examples in Chapter 2, each path in a symbolic execution tree represents a possible prefix of an execution path. The symbolic execution tree is constructed by adding new nodes and creating new paths based on the messages generated during test runs. The types of messages a test input selector can receive are shown in Table 4.1 as a summary from Chapter 3. Assume that a node has been

```
List l1 = input();
List l2 = input();
int x   = input();

if (x == 5) {
  if (l1 != null)
    if (l2 != null) {
      l1.value = l2.value;

      if (l1 == l2)
        print(l1.value);
  }
}
```

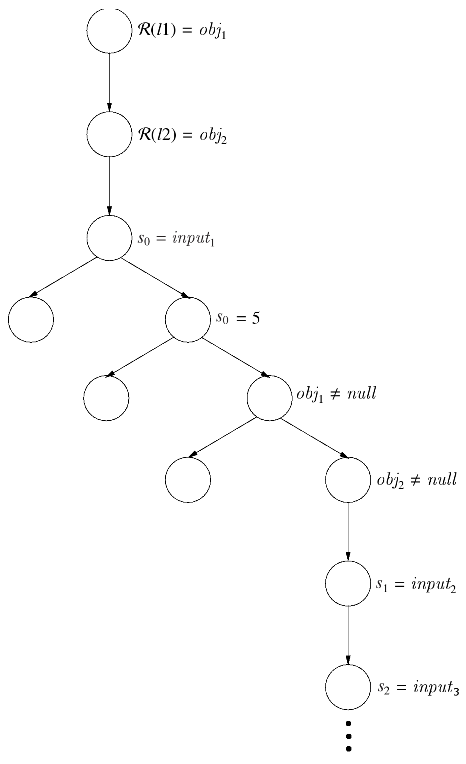


**Figure 4.2:** Symbolic object example

selected from the symbolic execution tree as the current node that is to be expanded. As a basic principle, all received assignment messages create a single child node to the current node being expanded and the assignment in the message (*e.g.*, $s_0 = input_1$ or $s_5 = s_2 \times 8$) is set as a constraint for all the paths that contain the newly added node. A branching message creates two child nodes and sets the constraint in the message as a constraint for the first node and a negation of the constraint for the the other node. The child node that corresponds to the path that the concrete execution is following is chosen as the node that will be expanded by future messages and the other is marked as unvisited.

Goto limit, method call limit and error messages denote that the current path is not explored any further and the current node that was to be expanded is marked as finished. If both children of the parent of this node are marked as finished, the parent is set to be finished also. If the used search strategy does not need information about the already visited subtrees in the symbolic execution tree, it also possible to delete unnecessary nodes from the tree. This can be done when a node is marked finished and it does not have any children or the child nodes have been marked as finished.

For symbolic execution that uses only primitive data types, the construction method described above is enough and a path constraint for an unvisited node can be formed by collecting all constraints on a path from root to the unvisited node. However, when symbolic objects are used with lazy initialisation the situation is slightly more complex. Consider the example program in Figure 4.2. Let us assume that the test input selector has generated inputs that will take the true branches on the first three if-statements and the false branch on the fourth. The partial symbolic execution tree that is constructed based on this test run is shown on the right side of the program code. Note that the symbolic list objects are initialised lazily when line 8 is executed. Assume now that for the next test run the test input selector wants to follow the same execution path as before with the exception that the execution is forced to take the true branch on line 10. We would like to start expanding the unvisited node corresponding to this path in the symbolic execution tree but if this is done, the resulting path would no longer represent exactly the symbolic events happening when the execution path is followed. This is because when $l1$ and $l2$ are set to point to the same object, the lazy initialisation of the latter object will not happen anymore as it is already initialised. Based on the first test run, it is expected that the third input value ($input_3$) will be used during lazy initialisation but on the second test run, the third input value may be assigned at some later point if the program continues beyond the code presented in our example. This can lead to giving input values to the test executor in a wrong order. Moreover, in the general case the lazy initialisation process may add new constraints and branches to the symbolic execution tree depending on whether the user has implemented a custom initialisation method and this can cause further inconsistencies to the symbolic execution tree.

To avoid this problem a symbolic execution path must branch into separate paths when different objects are given as inputs at the same input location. This approach introduces two questions that need to be answered. Firstly, where in the symbolic execution tree the branching to two separate paths should occur and secondly, what branches must be created based on the collected constraints.

Let us concentrate on the first question and assume for a moment that all the branches that need to be created due to different object inputs are known. One possible place to make a branch in the symbolic execution tree is the point where a program gets an object as input. As this is the location where a program can start to use the concrete input object it seems natural to create a branch at this point for each different object that are given as input. A partial symbolic execution tree of the program in Figure 4.2 that has been constructed using this approach is shown in Figure 4.3. Note that branching points have been specifically marked with nodes filled with grey colour on the tree and for clarity these nodes are allowed to have more than two children although an equivalent branching could be done with binary trees by using additional nodes.

Although with a symbolic execution tree like shown in Figure 4.3 it is possible to avoid the inconsistency problems when the same input object is used at different input location, there is a downside in creating such trees. Consider the nodes in the tree marked with number 1. The execution paths leading to those nodes are in fact the same in all cases (*i.e.*, the same statements are executed to reach them). Imagine that the program in Figure 4.2 continues after the line 13 and the program does not use the objects $l1$ and $l2$ after the example code snippet. This results in the fact that all the subtrees of the nodes marked with number 1 are identical and therefore exploring all the paths shown in Figure 4.3 leads to exploring the same program behaviour multiple times. Exploring these identical executions can be very expensive especially if the subtrees are large.

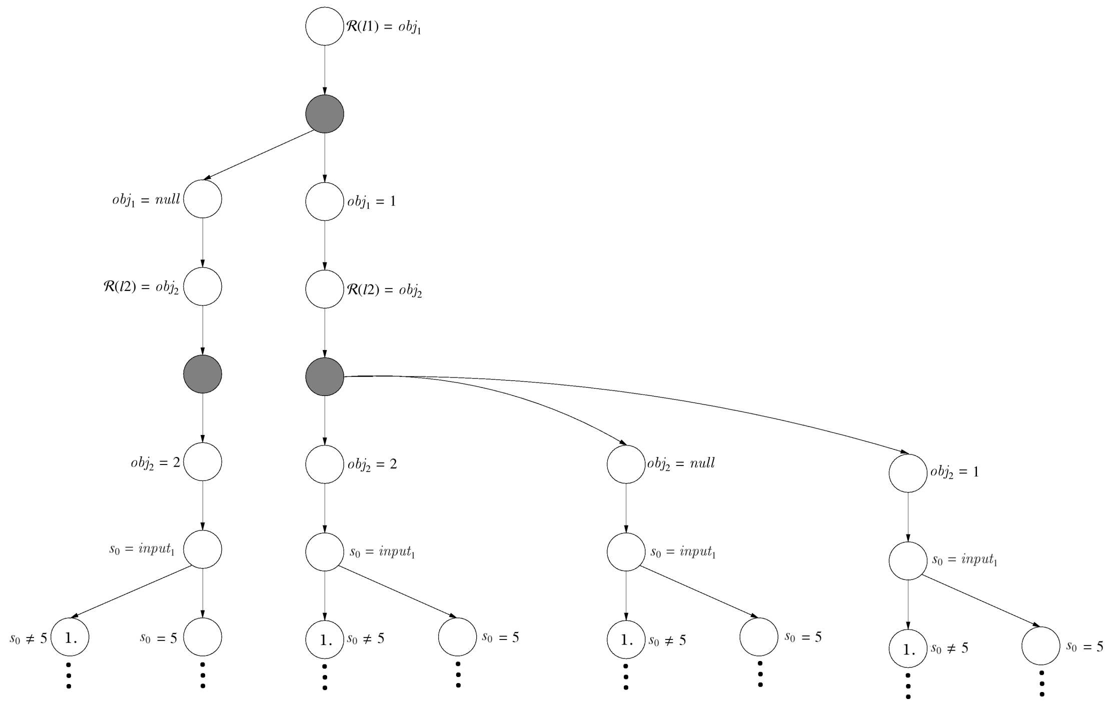


**Figure 4.3:** Symbolic execution tree with branching for different object inputs

Note that using the approach taken in jCUTE where lazy initialisation is not used has the same problem. In jCUTE the input objects are set to be null in the first test run that uses them and initialised to be non-null for the following test runs. Due to initialising the input object at the input location the symbolic execution effectively branches at the same locations as in Figure 4.3.

It is, however, possible to improve the situation by changing the location where the branching points are created. Although it is true that the program receives a concrete input object when an input statement is executed, it is also true that the object is not used before one of its fields or methods is accessed. Therefore we propose that the branching points are added to the symbolic execution trees right before the lazy initialisation points because input objects are not used before lazy initialisation. This way it is possible to take advantage of the fact that there can be statements between an input location and the point where an input objects is started to being used such that these statements create execution paths that are not needed to be explored multiple times.

An example of this improved approach is shown bellow but before that the question of how to create symbolic execution trees with the discussed kind of branching is addressed. We propose the following approach to constructing the necessary branches. When a test input selector receives a lazy initialisation message, a branching point is added to the current symbolic execution path. From this branching point a single outgoing branch is created where the logical address of the object to be initialised is fixed to a value that has not yet been used on the current path (*e.g.*, setting a constraint $obj_2 = 2$). The symbolic execution continues normally after the branching

```
CHECKALTERNATIVE($n$)
 1  if ($n.constraint$ is of type $obj_i = obj_j$)
 2      $m = n$;
 3      $l =$ not found;
 4      $t = null$;
 5      while ($m \neq T.root \wedge l =$ not found)
 6          if ($m.constraint \in \{obj_k = a \mid k \in \{i, j\}, a \in \mathbb{N} \setminus \{0\}\}$)
 7              if ($t = null$)
 8                  $t = m.parent$;
 9                  $x = k$;
10              else
11                  $l = a$;
12          $m = m.parent$;
13      if ($m \neq T.root$)
14          if ($t$ does not have a child with constraint $obj_x = l$)
15              Add a new child node to $t$ with constraint $obj_x = l$;
16              Mark the new node unvisited;
```

**Figure 4.4:** Adding branches for different input objects

point has been created. Consider now that in the subtree after the point where the logical address of the object was assigned is an object constraint that contradicts the assignment (*e.g.*, in the symbolic execution path it is fixed that $obj_1 = 1$ and $obj_2 = 2$ and there is an object constraint $obj_1 = obj_2$). Giving a path constraint with such assignments and an object constraint to a constraint solver results in getting an answer that the path condition in unsatisfiable. However, the path condition might be unsatisfiable because of the fixed assignments made at branching points and it is possible that with different assignments the path condition becomes satisfiable. If this is the case, new branches are needed to be created to the branching locations. To do this our tool executes a CHECKALTERNATIVE method whenever an unsatisfiable path constraint is encountered as shown in Figure 4.1. The method itself is shown in Figure 4.4.

To determine whether a new branch is needed to be added to the symbolic execution tree when a path constraint is found to be unsatisfiable, CHECKALTERNATIVE first checks if the last constraint of the path constraint is an object constraint that compares two input symbols to each other (line 1). If this is not the case, it is safe to say that the path constraint is truly unsatisfiable. The reason for this is that the last constraint corresponds to a branch in the program that was not followed by a concrete execution that caused the node to be added. This means that the path constraint without the last constraint is quaranteed to be satisfiable. As only object constraints affect the values of objects, there is no need to proceed further if the last constraint is not an object constraint. Additionally, the checking can be stopped if the object constraint requires an input symbol to have a specific logical address as the only cases where these kind of constraints are constructed are when CHECKALTERNATIVE is called or when an object is required to be a null reference. If an object is required to be a null reference and lazy initialisation (and a commitment to a specific logical address) of that object has already been done, it is not possible that the object can be null as at least one of its fields has been accessed. To catch null pointer exceptions when no object constraint requires an object to be null, it is checked at the lazy initialisation points if the collected path constraint so far allows the object to be null. If this is the case, inputs that will necessarily cause a null pointer exception can be generated.

If the last constraint is of the correct type, the symbolic execution path is then searched (lines 5-6) for branching points where input symbols in the last constraint are assigned their logical addresses. If branching points cannot be found for both of the input symbols (*i.e.*, the search was not stopped before the root of the tree as checked in line 13), the path constraint is unsatisfiable as the commitments to specific logical addresses cannot in this case be the cause of the unsatisfiability. If the branching points are found, a new branch is constructed at the branching point that was found first (*i.e.*, located deeper in the symbolic execution path) with a commitment to the same logical address as used in the other located branching point unless such branch has already been created. The node corresponding to this newly added branch is then marked as unfinished.

**Example 6.** Let us look at how the symbolic execution tree of the program shown in Figure 4.2 is constructed by using the approach described above. Figure 4.5 shows the first test run on the left side of the figure, which is the same as the one depicted in Figure 4.2. The only difference is that new branching points shown with dark grey colour are added to the tree with fixed assignments $obj_1 = 1$ and $obj_2 = 2$. Let us assume that the unvisited node with the object constraint $obj_1 = obj_2$ is selected to be extended next. The path constraint corresponding to this node is unsatisfiable because the fixed assignments contradict the object constraint in the unvisited node. Because of the unsatisfiability the CHECKALTERNATIVE method is called and it locates the two branching points and adds a new node with a constraint $obj_2 = 1$ to the branching point located deeper in the symbolic execution tree. This new node is marked unvisited and can be explored instead of the node that was originally selected.

The second test run, shown on the right side of Figure 4.5, extends the newly added node by requiring both of the object inputs to have the same logical address. Note that the second test runs creates an unvisited node with object constraint $obj_1 \neq obj_2$. This is again unsatisfiable with the used object assignments but as the last constraint is not of the type $obj_i = obj_j$, CHECKALTERNATIVE determines that this branch must be truly unsatisfiable (as different input objects are always initialised to be different unless required otherwise by a branchin statement, such input must have already been used during earlier test runs).

When the trees in Figure 4.3 and Figure 4.5 are compared, it can be seen that the branch where $x \neq 5$ is explored only once in Figure 4.5 instead of four times as in Figure 4.3. Although the approach described here helps avoiding some test runs that exercise the same behaviour, there is still room for further improvement. This is because the CHECKALTERNATIVE method creates new branches at the lazy initialisation

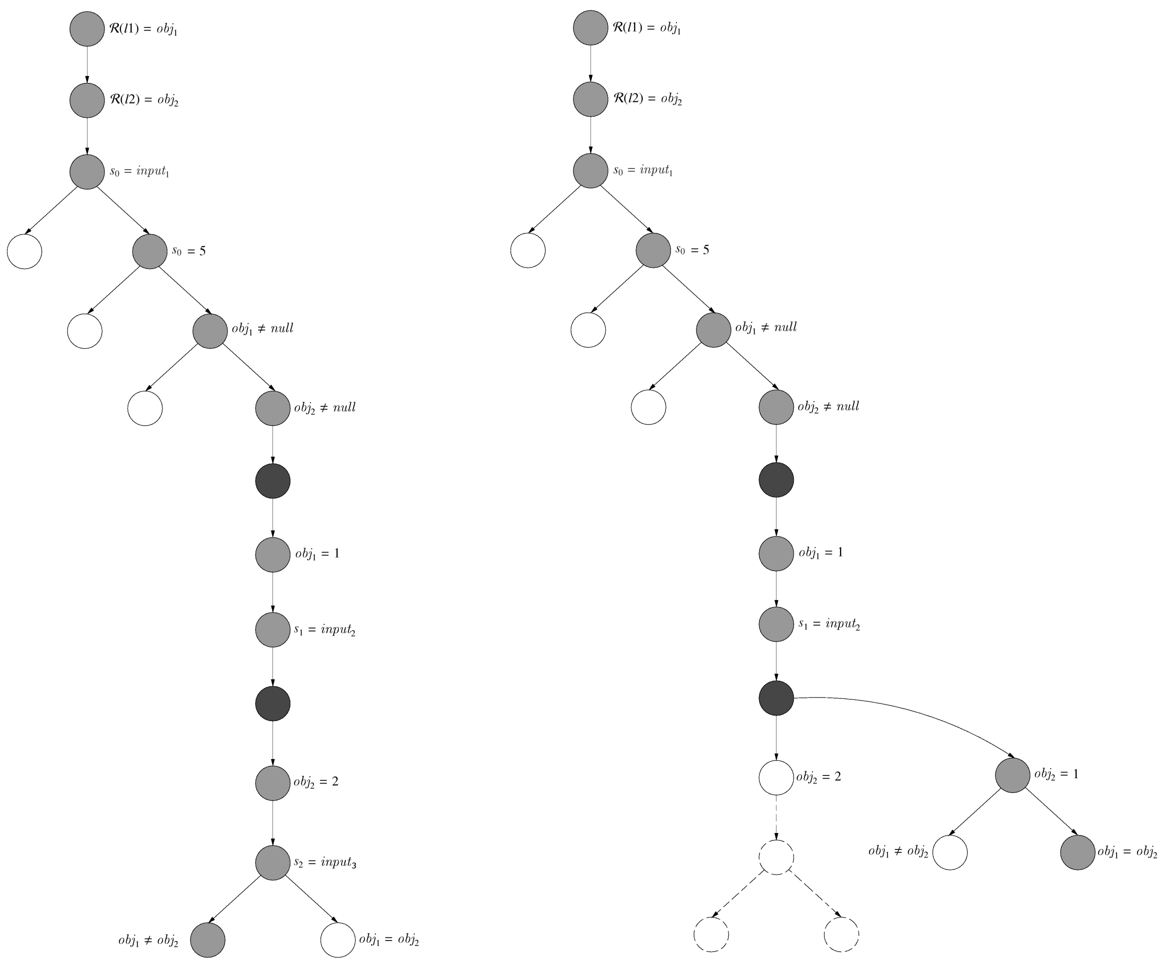


**Figure 4.5:** Example of additional branching for different input objects

point but does not take into account the execution that happened after the initialisation on the test run that created the new branch. In other words, each time a new branch is added to a branching point, the whole subtree after that point is explored for each of the fixed assignments. These subtrees can containt execution paths that are the same as in other already explored subtrees.

Note also that the described way of creating a symbolic execution tree and path constraints is not necessarily optimal. Currently the symbolic assignments are considered as local constraints and as each different symbolic value is represented by an unique symbolic identifier, the path constraints can contain a large number of variables that the constraint solver must process (*e.g.*, $s_0 = input_1 \land s_1 = s_0 + 5 \land s_1 > 0$ instead of $input_1 + 5 > 0$). One possibility is to remove the intermediate variables by writing open the path constraints so that they contain only input variables. It is also not necessary for the test executors to report assignment events as it would be enough to only send local constraints in the open written form to the test input selector. As the constrains in that form can become large, they should be simplified whenever possible (this process is discussed later in Section 4.3.3). If the local or path constraints are processed to the simplified open written form by test executors, this processing will

```
 1 int example (int x, int y)
 2 {
 3     if (y > x)
 4         if (x > blackBox(y))
 5             ...
 6         else
 7             ...
 8     else
 9         ...
10 }
```

**Figure 4.6:** Example of execution failing to follow the correct path

be done for each constraint no matter if they are used or not due to the number of execution paths being so large so that the test generation process is terminated before all of them are explored and therefore before all the path constraints are solved. On the other hand, if the test input selector processes constraints, it can be done on demand to avoid doing unnecessary work but it would also mean that the task is moved to the selector that already must coordinate with all test executors that are running in case that many of them are run concurrently. Evaluating different alternatives and improving the current way of constructing and storing path constraints is left for future work.

### 4.1.2 Failing to Follow a Predicted Execution Path

When a new test run is started, one of the nodes in the tree corresponding to an unexplored branch is selected and the tree is expanded from that node onwards based on the messages the test executor sends as explained in the previous subsection. Before the tree can be expanded, it is necessary to check that only those messages are used that are received after the test run has reached the point where the execution of an unexplored part of the execution path has started. It is also necessary to check that the test run will in fact follow the predicted execution path. The reason why this check is needed is that our tool approximates black box methods and operators not supported by the constraint solver with concrete values and also because it does not create exact path constraints with aliasing as shown in Chapter 3.

**Example 7.** Let us consider the method in Figure 4.6. Assume that the method blackBox is uninstrumented and returns the same value as it gets as an argument. Because the method is uninstrumented, it will only return concrete values. In other words, the symbolic value associated with variable $y$ is stripped away by the method. Now if the example method is tested with input values $x = 5$ and $y = 10$, the program will end up executing code from line 7 onwards and at the same time get a path constraint $y > x \land x > 10$ that is assumed to correspond to inputs that lead to executing line 5. A possible solution of $x = 11$ and $y = 12$ to this constraint will, however, end up also executing line 7 and if this deviation from the expected path is not noticed, the test input selector ends up expanding a wrong node in the symbolic execution tree.

To make sure that a test run follows the expected execution path the bit-vectors corresponding to the branches taken during concrete execution as explained in Chapter 3 are used. When a node corresponding to an unvisited branch is added to the symbolic execution tree, a bit-vector containing the path to the node is stored to it. When a node is selected to be expanded, the test input selector checks whether the start of the bit-vector created by the test run matches the stored bit-vector. If it does, the node can be expanded with messages received after the bit-vectors match. Otherwise the test run has chosen a different branch at some point of the execution. In this case the target node is set to be finished and the user is notified that the tool has failed to do precise symbolic execution and some possible execution paths may be left untested as a result.

## 4.2 Search Strategies

As each test run can potentially add multiple new branches that can be explored to the symbolic execution tree, the test input selector has the possibility to choose which of these unexplored branches is to be tested next. When the aim is to explore all of the execution paths of a given program, the order in which the execution paths are tested makes little difference[1] except that the time to find the first error might differ. However, when the number of execution paths is too large in order to explore them all within a feasible time limit, different search strategies and heuristics will perform differently. In this section we will look at the three search strategies that have been implemented in our tool.

### 4.2.1 Depth-first and Breadth-first Searches

One of the simplest ways of selecting an unvisited node from the symbolic execution tree is to use classical depth-first (DFS) or breadth-first (BFS) searches. In both of these cases the symbolic execution tree is traversed according to the search strategy and the first unvisited branch located is selected and new input values are computed by solving the path constraint associated to that branch.

DFS has the positive side that when only one test executor is run at a time, the search strategy will systematically explore one subtree of any node before exploring the other and once a subtree has been fully searched it can be deleted from memory. This makes DFS very memory efficient. The fact that one subtree is systematically explored before other possible execution paths is also a weakness of DFS as the strategy will get stuck exploring only a small local part of a program under test if it has a large enough number of execution paths so that only a small amount of them are tested. Another

[1]If some execution path reduction methods are used, their performance may also depend on the search order but this fact is not discussed further here.

downside with DFS is that as it aims to select nodes that are deep in the symbolic execution tree, the path constraints for these are longer than for the nodes closer to the root node and as such more difficult for a constraint solver to solve.

BFS on the other hand requires most of the symbolic execution tree that has been generated during test runs to be kept in memory but at the same time avoids many of the weaknesses of DFS. The strategy does not get stuck in the same way as DFS does and it also aims to solve the easiest path constraints first. One downside with BFS is that if the program, for example, is a parser and it does some input filtering as it expects to read input strings with correct syntax, BFS concentrates its effort on exploring these early paths and finds all the possible ways of creating incorrect inputs to the system. This is usually uninteresting if the aim is to test that the parser works correctly with correct input values. In otherwords, BFS explores systematically the early branches in the control flow of a program and for this reason often misses bugs deep in the symbolic execution tree if the whole tree is not explored.

Both of these strategies search the symbolic execution tree in a fixed order without taking the structure of the program under test into account which could increase the possibility on finding errors on large programs. Also both of these strategies work in their intended way only when no more than one test executor is running at a time. It is of course possible to use these strategies with many test executors running concurrently by normally searching the symbolic execution tree (which might get modified during the search). This requires that each branch that is being expanded currently by some test executor is marked as such so that they cannot be given to be tested by another test executor. With DFS this means that the memory efficiency advantage is somewhat diminished as the subtrees of nodes might not be strictly tested in a fixed order. In the BFS case running tests concurrently does not introduce any new disadvantages.

### 4.2.2 Random Priority Search

The third search strategy in our tool, which is called a random priority search, is designed to be used easily with a varying number of test executors running concurrently and to avoid the localisation problem of DFS and the preference of BFS to concentrate on covering the branches near the root of the symbolic execution tree first. In this search strategy each new unvisited branch that is added to the symbolic execution tree is given a priority value randomly on some predefined value range and when new input values are required the branch with the highest priority is selected to be tested next. To be able to get the branch with the highest priority without searching the whole symbolic execution tree, a priority queue is maintained as an auxiliary data structure that contains pointers to the branch nodes.

Downside with random priority search is that it can ignore some branches if they get low priorities. For example, if at the first branch added to the symbolic execution tree the unvisited branch gets the lowest possible priority, the whole subtree of the branch that was taken during the first test run is explored before testing any of the symbolic execution paths on the other subtree. One possibility to address this problem is to periodically reassign new random priorities to the unvisited nodes. This requires updating the whole priority queue which can be expensive if the number of unvisited nodes is large.

Like the DFS and BFS searches, the random priority search in its basic form does not take the structure of the program under test into account. To make the search less random, a variant of the random priority search has been implemented where the range of the priorities assigned to unvisited nodes depend on whether the node is created due to a branching statement on a specific code line that has already been executed in one of the previous test runs or not. Distinguishing one branching statement from another can be done by giving unique static ID to each branching statement during instrumentation and then reporting the ID whenever a local or object constraint is created. If the branching statement was not executed before, we know that executing the unvisited branch of this statement will lead to exploring a previously unexplored branch in the control flow graph of the program. If such branches are given a higher probability to get a better priority, the search strategy will be guided towards getting better branch coverage before exploring other execution paths. By adjusting the ranges of possible priorities for the two cases, the user can affect how greedily the search aims to improve branch coverage.

Currently our tool does not collect during an execution any other information that could be used to adjust the priorities in addtition to the branching statement IDs. We plan on investigating new ways of getting useful information from the execution or the structure of the program so that we can use the priority approach to guide the testing to find errors more efficiently.

### 4.2.3 Combining Search Strategies

If the data structure representation of the symbolic execution tree is compatible with multiple search strategies, it is possible in our tool to use them all together in the same testing process. In other words, when test executors request new inputs, it is possible to choose a search strategy that will be used regardless of the search strategy that had been used previously.

In [26] it was discussed that a search strategy may perform better with some problems and worse on others when compared to another search strategy. This gives the idea that our ability to find errors is not so strictly dependend on whether a single strategy performs well on the current problem or not if multiple different strategies are combined to a new meta strategy as is done in [29]. A framework for allowing different strategies to communicate with each other was also presented in [26]. This allows strategies to guide one another to interesting paths and take advantage of each others strengths in different situations. For example, a random search could find an interesting branch and DFS search would then be started to explore the subtree of this branch in a more thorough manner. We have not yet implemented any such strategies that can do this kind of communication but one possible direction for future work is to investigate what makes an execution branch interesting in our symbolic execution context and to develop more sophisticated search strategies.

### 4.2.4 Reporting Errors

As the aim of testing is to search for errors in a given program, the errors found during test runs are reported to the user. When an error is found, the input values used to reach the error state are stored so that the same execution can be repeated at a later time if this is desired. Naturally, if the aim is to generate test inputs for a later use, the concrete input values for all distinct execution paths can be stored and given to the user.

If the program contains an error that can be reached by multiple different execution paths, our tool will report an error for each of these paths unless the user selects a limit to the number of errors after which the search is terminated. This can in worst case lead to reporting a large number of errors because of one bug in the implementation. For example, if a program throws an exception if a null element is added to an input data structure, our tool might generate all possible variants of the data structure where the null element is added. To make it easier for the user to analyse the found errors, it is possible to give inputs for the shortest execution in respect to the number of symbolic branching statements encountered during execution by sorting the found errors by the depth in the symbolic execution tree where the error happened. Note that this does not guarantee that the concrete execution to the error statement is the shortest one as we do not know the number of statements using only concrete values that will be executed in any execution path.

## 4.3 Solving Path Constraints

It has been described so far how an unvisited branch is selected from the symbolic execution tree and how the path constraint for that branch in obtained. In this section a closer look is taken at how the input values for new test runs are computed with the help of a constraint solver.

### 4.3.1 Local and Object Constraints

To obtain inputs from a path constraint it is first divided into two parts that are solved separately. The first part consists of all the local constraints and the second part of all the object constraints. Solving the first part is simple. The conjunction of all local constraints are given to a constraint solver and if this conjunction is satisfiable, the concrete values for each symbol in it are obtained. From these symbols the input symbols and their respective values are picked (the other symbols are the intermediate identifiers $s_0, s_1$ and so on) and given to the test executor.

Solving the second part of the path constraint requires more steps. The conjunction of object constraints is first given to the constraint solver that consideres the symbols in it to be integer variables. If the conjunction is unsatisfiable, so is the whole path constraint and if it is satisfiable, the constraint solver could give some concrete integer values to the object symbols that can be considered to be the object identifiers.

However, the satisfying assignment from the constraint solver is not used. The reason for this is that there are additional requirements for the identifier values that are not expressed explicitly in the path constraint. For example, consider a path constraint $obj_1 \neq obj_2 \wedge obj_3 = obj_4$. One possible satisfying assignment is $obj_1 = 1, obj_2 = 2, obj_3 = 3$ and $obj_4 = 3$ but the constraint is also satisfied if $obj_1 = 3, obj_2 = 0, obj_3 = 3$ and $obj_4 = 3$. There are two problems shown by this example. Firstly, we have an assumption that symbolic objects will be initialised as null references only is there is an object constraint requiring this. In the second solution the assignment $obj_2 = 0$ breaks this assumption. This has the effect that as we do not have symbolic values associated with null input objects, some possible branches could be left out of the symbolic execution three if this input object is compared with non-symbolic objects or null references. Further more, there are no quarantees that the constraint solver used will always return the same concrete values for a same kind of constraint. If on the first test run the constraint solver gives an assignment $obj_2 = 0$ and on the second test run assignment $obj_2 = 2$, the test runs might follow different execution path prefix if some branches were left out during the first test run as explained above.

The second problem is that we also assume that two symbolic objects will be the same only if there is a requirement for this in the path constraint. In the second assignment the first, third and fourth input objects are set to be the same. This could for example lead to a case where a system is only tested with an input linked list that has only one element that points back to itself. This, of course, is a valid input structure but as in this case only one concrete input object is created, our tool might not be able create other kind of inputs as all the input objects now have the same symbolic value.

Because of the problems discussed above the constraint solver is used only to check if the constraint is satisfiable and if it is, identifiers are assigned to the object symbols in the following way. Our tool first builds an equivalence graph based on the object constraints. This is done by adding each object symbol to the graph as nodes with *null* as a special node and adding an undirected edge between nodes if the there is an object constraint that requires the corresponding object symbols to be equal. Because the constraint is satisfiable, there is no need to worry about disequalities that could make the constraint unsatisfiable, that is, all object constraints with disequalities are ignored while constructing the graph. The graph constructed this way divides the node into equivalence classes. To the get the identifier values for the symbolic objects, one node is picked from the graph, all nodes that are reachable from that node are collected and all the object symbols corresponding to these nodes are given the same value. If the set of object symbols collected this way is $N$, the value given to all symbols in $N$ is determined in the following way. If the path constraint contains an assignment for one of the object symbols in $N$, the value in that assignment is used. Otherwise the value is a logical address that has not been used before on the symbolic execution path in question. After the values are given, the nodes corresponding to the symbols in $N$ are removed from the graph and the process is repeated until the all the nodes have been removed from the graph. The object identifiers computed this way are then given to the test executor.

### 4.3.2 Solving Constraints Concurrently

The alert reader might already have noticed that there is a problem on how to solve path constraints when the tool is used with multiple test runs executing concurrently. If the test input selector solves path constraints when input values are required to start a new test run, all the calls to the constraint solver happens in one centralised place and this can become a performance bottleneck. To avoid this problem a way to distribute constraint solving to different computation nodes is required. We have considered two alternatives for this: the testing system can be connected to a pool of dedicated constraint solvers running on different nodes or the path constraints can be given to the test executor to solve before it starts the actual test run.

The first approach has a problem with load balancing. Ideally we want to start a test run on one node immediately after the previous test run has ended. If the pool of constraint solvers is too small, the test runs have to wait until a solver becomes available. On the other hand, too large a pool will only waste resources that could be, for example, used for executing test runs. For these reasons we have used the second approach in our implementation where the test executors solve the path constraints on demand. This partially blurs the division of the testing system to separate test input selector and test executors but solves the load balancing problem and makes the testing system easier to set up for the end user as there is no additional pool of solvers involved.

### 4.3.3 Optimisations

The approach described above on how path constraints are constructed and solved can be improved in various ways. We have implemented two optimisation for solving path constraints. The first is called *fast unsatisfiability check* [27] where before giving a path constraint to the constraint solver it is checked if the last constraint is a syntactic negation of any of the preceding constraints. It if is, we can be sure that the path constraint is unsatisfiable without having to call the constraint solver. Checking the last constraint this way is based on the observation that the path constraint without the last constraint must be satisfiable as it has been used to compute the input values for the test run that caused the last constraint to be added.

The second optimisation is to store concrete input values used during current test run to unvisited branches in the symbolic execution tree. If the branch is created due to adding a local constraint, then the current object identifiers are stored and when adding a object constraint, the concrete input values are stored. Now when a path constraint must be solved, we need solve only either the local or object part and use the concrete values from the previous run for the other part. This can be done because, as discussed above, only the last constraint can make the path constraint unsatisfiable and as the local and object parts are solved separately, the last constraint can affect only one of them.

The first optimisation is severely limited in our current implementation due to the fact that we always introduce new symbolic identifiers (*e.g.,* $s_0$ and $s_1$) when a symbolic value changes. For example, if we have a path constraint ending with $s_7 > 0 \wedge s_7 \leq 0$, we can use the optimisation. But if the variable having the symbolic value $s_7$ is first summed with some value ($s_8 = s_7 + c$) and then subtracted the same value ($s_9 = s_8 - c$), we have a new symbolic identifier for this value even though it represents a same value. With these symbolic values the fast unsatisfiability check fails on a path constraint ending $s_7 > 0 \wedge s_9 \leq 0$. It would be possible to improve this situation by not creating a new symbolic identifier each time a value changes but to simplify the symbolic values (*e.g.*, $s_0 + 5 + 2 = s_0 + 7$). Simplifications based on folding constants and symbols is just one technique that can be used. In [29] additional simplification approaches based on using BDD [7] representations of logical connectives and hash-consing [14] among others as discussed.

In CUTE [27] the idea of our second optimisation is taken even further by an optimisation that allows path constraints to be solved incrementally. Note that taking advantage of incremental constraint solvers is difficult especially with search strategies such as the random priority search as the constraints solved consecutively can have few constraints common in them. The approach used in CUTE can be used without an incremental solver and it can be utilised with any search strategy. The optimisation based on the notions of dependency between different constraints in a path constraint. According to [27], two constraints $c$ and $c'$ are dependent if either

- $c$ and $c'$ have any common symbols in them, or
- there exists a constraint $c''$ in the path constraint such that $c$ and $c''$ are dependent and $c'$ and $c''$ are dependent.

As discussed before, if the last constraint is removed from the path constraint $C$, it is quaranteed to be satisfiable. We can now go though all the constraints in $C$ and collect the ones that are dependent with the last constraint. The conjunction of these constraints is then given to the constraint solver and in case it is satisfiable, the satisfying assignment is augmented with the concrete values used by a previous test run to obtain all the input values. By using this optimisation, it was reported in [27] that the constraints given to the constraint solver were reduced on average to one-eight the size of the original path constraint on many cases.

# Chapter 5

# Implementation

In this chapter the implementation details of our tool that is based on the methods described in Chapters 3 and 4 are discussed. The limitations our tool has that the users should be aware of are also discussed.

## 5.1 Structure of the Testing System

The structure of our tool is shown in Figure 5.1 and it can be seen as consisting three main parts: the instrumenter, the test input selector and the test executors. The instrumenter is based on a tool called Soot [30], that can be used to analyse and transform Java byte code. Before a program is given to the instrumenter, the input locations in the source code are marked so that the instrumenter knows how to transform the code. Our tool provides a static class that is used for this in the following fashion:

- `int x = Symbolic.getInteger()` is used to get an int type input value for a variable x, and
- `List l = Symbolic.getObject(“List”)` indicates that an object l is an input object.

Our tool has support for all primitive data types in Java as symbolic inputs with the exception of float and double data types. Support for these missing types is planned to be added in near future.

After the input variables have been marked in the source code, the program is given to the instrumenter that transforms the code into an intermediate representation called Jimple and adds the statements necessary for symbolic execution into it. When the instrumentation is finished, the code is transformed into byte code that can be run over a standard Java Virtual Machine (JVM).

The instrumented program can be seen as a test executor. The test executors communicate with the test input selector to report the collected constraints and to receive

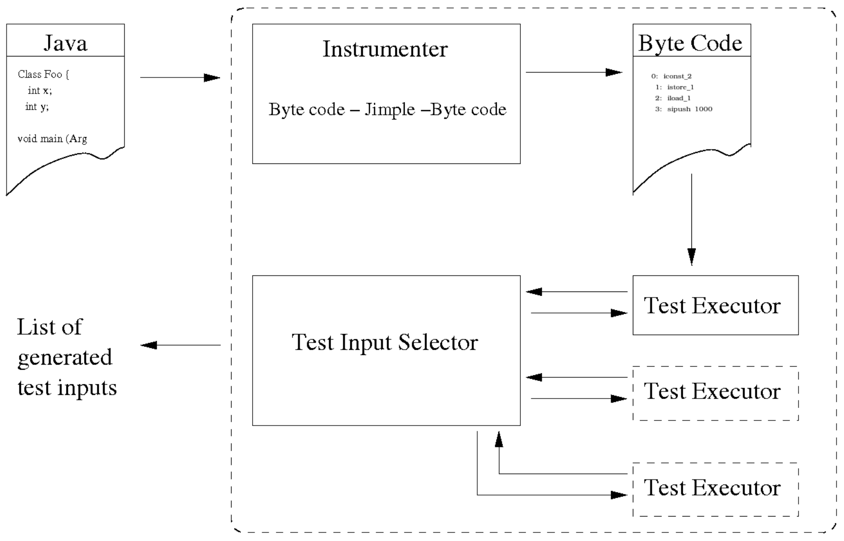


**Figure 5.1:** Structure of the testing system

new input values. This communication is implemented by using TCP/IP connections. The test input selector builds a symbolic execution tree to maintain the symbolic information about different execution paths. Our tool uses Yices [11] as a constraint solver but it is easy to modify the tool to add support for other constraint solvers as well because the solvers are called though a single class that separates them from the rest of the implementation.

The system can be also seen as a client/server architecture, where the test input selector acts like a server and the test executors as clients. As different test runs do not depend on each other, it is possible to have multiple test runs executing concurrently and reporting to the test input selector. This allows the tool to take advantage of multicore processors and networks of computers.

As the test input selector and test executors are separate, it is possible to use a different kind of instrumentation, possibly even for a different programming language, to obtain a runnable test executor and still use the same test input selector provided that the new test executors use the same communication protocol with the implemented test input selector. The same goes also the other way around, it is possible to replace a test input selector with an alternative implementation without the need to modify how the test executors are constructed. The implemented tool is also designed to be modular with respect to the search strategies. Adding new strategies for the test input selector requires only writing classes that implement the symbolic execution tree and the functionality that operates on the tree based on the messages received from the test executors. Rest of the tool needs little modification for new strategies except in the case that additional information is needed to be collected during test runs which can require additional instrumentation.

## 5.2 Limitations

The main limitations in our tool concern the automatic instrumentation process. In order for the dynamic symbolic execution to be a worthwhile approach to generating test cases, at least the key parts the program under test must be instrumented. Otherwise our ability to construct a reasonably extensive symbolic execution tree is severely limited as the symbolic values will be approximated with concrete values. This problem becomes especially clear with programs using the standard libraries provided in Java Development Kit (JDK) as a part of them are restricted in a way that a modified versions of them cannot be used with the standard JVM. Consider, for example, a program that places input values to an object instance of a String class. If the String class cannot be instrumented, all the symbolic information associated with the inputs are lost and the use of the String object will not result in constructing any path constraints.

Luckily Sun Microsystems has released most of the source code of Java and provides a public and freely modifiable version of the Java platform in OpenJDK project [25]. Investigating the use of of our tool with OpenJDK is planned for the future as it looks like a promicing solution to the problem with instrumentation of the standard libraries. Another possible solution is to implement a twin class hierarchy [13] of the standard libraries. In this case the problematic classes are duplicated with different names and these duplicates are instrumented while leaving the original classes untouched. The program under test would then be modified to use the alternative classes in place of the originals.

Also as our automatic instrumentation is based on the Soot tool, it is resticted to the Java version it supports. At the time of writing this work, Soot has an almost complete support for JDK 1.5.

# Chapter 6

# Case Studies and Discussion

In this chapter the use of our tool and its ability to find errors are illustrated by presenting case studies on testing various implementations of binary search trees and sorting algorithms. The suitability of dynamic symbolic execution for test generation will also be discussed based on the case studies and experience gained during the implementation of our tool. Some possible improvements to dynamic symbolic executions that have been suggested in the literature are also discussed.

## 6.1 Test Arrangements

The case studies in this work were made by running our tool on several data structures and sorting algorithms obtained from [32]. The data structures used were AVL tree and basic binary search tree. The implementations of these two data structures were tested by using a test driver that nondeterministically inserts to, removes from or searches from the data structure a key given as an integer input. For each test run this nondeterministic choice was repeated four times to modify or use the resulting data structure. As in [32] no implementation for operation that removes a node with a given key from an AVL tree is given, an implementation of this operation available at `http://cs-people.bu.edu/mullally/cs112/code/AvlTree.java` was used to complete the AVL tree data structure implementation.

The sorting algorithms used in the case studies were insertion sort, shell sort and quicksort. To test these algorithms a test driver was used that creates an array of size five and assigns an integer input as a value for each of the elements in the array. This array was then sorted with one of the selected sorting algorithms and afterwards it was checked that the sorting algorithm functioned correctly (*i.e.*, the values in the elements of the array are sorted). The method used for checking the correctness of the result of the sorting algorithms is shown as Method 1 in Appendix A.

The used implementation of insertion sort compares the values of elements in a given array by using a method that checks whether one value is greater than, less than or equal to a second value. In insertion sort it is enough to know whether one value is less than some given value and therefore an additional experiment was made where

| Test Case | RT 1 | RT 2 | Paths | Unsat | Fast unsat |
|---|---|---|---|---|---|
| AVL tree | 4m 2s | 49s | 1233 | 1086 | 98,9% |
| Binary search tree | 4m 1s | 39s | 1233 | 1448 | 99,7% |
| Insertion sort 1 | 2m 50s | 28s | 541 | 2316 | 27,4% |
| Insertion sort 2 | 34s | 6s | 120 | 480 | 50,0% |
| Shell sort | 2m 55 | 29s | 541 | 3238 | 45,5% |
| Quicksort 1 | 2m 51s | 28s | 541 | 2316 | 27,4% |
| Quicksort 2 | 27m 45s | 4m 33s | 4683 | 24833 | 27,4% |
| Quicksort 3 | 17m 8s | 2m 47s | 4683 | 1420 | 0% |

**Table 6.1:** Case studies

the comparison method was replaced with normal less than operator that gives only true or false as a result instead of the three possibilities in the original version. With quicksort two additional experiments were made. In the first one the size of the array was increased to contain six elements and in the second one the array size was also set to be six but the check that the sorting was done correctly was removed.

The experiments were run with two different computer setups. In the first setup one computer was used to run a single test executor at a time together with a test input selector. In the second setup the test input selector was run on one dedicated computer and six other computers were used to run concurrently one test executor per computer. The computers used in the experiments had Intel Core 2 Duo processors running at 1.8 GHz together with 2 GB of RAM. Random priority search was used as the search strategy for each of the test runs.

## 6.2 Test Results

The results of applying our tool to the cases described in the previous section are shown in Table 6.1. Insertion sort 1 is the unmodified version of the implementation given in [32] and Insertion sort 2 is the version where the value comparison is changed. Quicksort 1 sorts an array of size five and in Quicksort 2 the size is increased to six. Quicksort 3 is the experiment without the correctless check. RT 1 and RT 2 denote the running times of testing using one computer and seven computers respectively. Paths denotes the number of test runs needed to explore all the paths in the symbolic execution tree and unsat tells the number nodes in the symbolic execution tree that had unsatisfiable path constraints. Fast unsat gives the percentage of how many of these unsatisfiable path constraints were identified as unsatisfiable by using the fast unsatisfiability check described in Chapter 4.

Our tool did not detect errors on any of the implementations which was expected as they are all text book examples. From the results in can be seen that the testing can be efficiently done by using multiple test executors that run concurrently. On these experiments dividing the work to seven computers is approximately five to six times faster than in the case where only one computer is used.

When Insertion sort 1 and Insertion sort 2 are compared, it can be seen that the way how two values are compared to each other has a noticeable effect on the number of possible execution paths. Whether the comparison can return two or three values has little effect on the efficiency of the sorting algorithm itself but our tool tries to compute input values that force the execution to explore all these possible return values. In other words, even small changes that do not affect the functionality of the program under test can make big differences to the size of the symbolic execution tree of that program.

By comparing Quicksort 1 and Quicksort 2 it can be seen that our tool does not scale well on large input arrays. In these experiments the reason for the blow up of execution paths is that in order to test all the possible execution paths in any sorting algorithm the number of test runs needed is factorial of the size of the input array. In other words, all possible permutations of the input array must be explored. In Insertion sort 2 it can be seen that the number of test runs is exactly 5!. In other cases the number of test runs in further increased due to the use of three valued comparison method that causes unnecessarily some permutations to be explored multiple times. This is also the reason why in many of the experiments the number of test runs is the same. Quicksort 3, on the other hand, shows that checking if the array has been correctly sorted does not add any new execution paths but is responsible for creating majority of the unsatisfiable path constraints. This is because the sorting algorithm works correctly and therefore it is impossible to create input values where the check would fail when any of the array elements are compared to each other. In other words, the check method used creates an unsatisfiable path constraint each time it compares two elements in the input array in order to check that they are in the correct order.

The fast unsatisfiability check functions well in these experiments and improves the running time as in many cases it is possible to skip a call to a constraint solver. However, these results can be misleading as all the programs tested use the input values directly and do not modify them. As our tool does not simplify the constraints as discussed in Chapter 4, it is expected that the fast unsatisfiability check does not perform as well in cases where input values are used together with, for example, summation and multiplication as in many cases it cannot be directly checked whether two constraints are negations of each other. In [27], where such simplifications are used, it is reported that the fast unsatisfiability check performs well on more varied kind of programs as well.

As the implementations of the selected data structures and sorting algorithms did not contain any errors according to our tool, the ability of the tool to detect errors was tested by changing the insertion sort algorithm to function incorrectly. The original algorithm is shown in Appendix A as Method 2 and the modified version as Method 3. The modification makes the sorting algorithm function incorrectly only when a specific input value is used. This kind of error was added because they are difficult to detect with random testing. As expected our tool was able to find input values that cause the correctness check to fail. Our tool reports that on the modified version there exists 1889 different execution paths and 586 of these contain an error. This can be seen as an example where a single bug can cause many errors to be reported. To test how fast our tool can detect a single error, the limit of number of errors to be reported was set to one and the insertion sort with the error was tested 20 times. In these experiments it took from three to 37 test runs to find the first execution path that causes an error. On average it took 16 test runs to find the first error. The variance in the number of test runs needed comes from the fact that the first test run is completely random and the random priority search can also search the symbolic execution tree in different order each time.

In addition to the tests described above an experiment was made to find out how much symbolic execution slows down the concrete execution without any instrumentation. This was done by adding timers to the start and end of the program code. To make the comparison, an uninstrumented version of insertion sort was executed with random inputs 200 times and the instrumented version was also run 200 times. On average the result was that in our tool doing symbolic execution together with the concrete execution was approximately nine times slower than executing the program without instrumentation. Note that this is only an approximation as the different versions most likely exercised different execution paths due to the random values used. The difference can naturally also vary between different programs as the amount of how much symbolic execution will be done varies from program to program. Nevertheless, it can be seen that the symbolic execution causes a clear slowdown to the running time of test runs. Note also that the time to solve path constraints is not included in this comparison as only the time to execute the program code was taken into account.

## 6.3 Discussion

Based on the case studies some general observations can be made. First of all, the test generation method described in this work suffers from a path explosion problem. This means that on large or even small but non-trivial programs the number of different execution paths can be so large that it takes too long to test them all. Therefore our tool does not scale well for large programs. A number of different techniques has been proposed in the literature to alleviate this problem. In hybrid concolic testing [23] dynamic symbolic execution is combined with random testing. In [24] a technique is introduced where low-level method calls are abstracted by replazing them with unconstraint input values (this can be seen as using a stub for the method that returns unconstraint values) and for the resulting execution paths it is checked whether a real execution path can be found through the method without abstraction. The aim in this technique is to concentrate on the program at hand and avoid generating tests for library methods that may already been tested or contain functionality not needed by the calling program.

Another approach to avoid generating all possible paths though library functions is to do symbolic execution compositionally. In [15, 1] various techniques are introduced where method calls can be expressed as summaries that are used instead of executing the method symbolically. The summaries can be seen as conjunctions of preconditions on method inputs and postconditions on method outputs so that when a summary is added to a path constraint it can describe multiple paths through the method using a single formula. In a way a summary can be seen as simultaneously following every path through the method and using a constraint solver to check that when an unvisited branch is wanted to be visited a path through the summarised method can be found. This way each method is needed to be executed symbolically only once and execution paths through the method that are not relevant to the rest of the program are not needed to be explored. The summaries can be constructed with separate unit tests or on demand while testing the main program.

Another factor in the path explosion problem is that some execution paths can have different prefixes but the postfixes exercise exactly the same behaviour. An example of this was seen in Chapter 4 when the program in Figure 4.2 was considered. As discussed at that point it is possible that large but identical subtrees are explored even when it is enough to explore only one of them. In [6] a technique called RWset is introduced that takes advantage of this fact and tries to prune paths from the symbolic execution tree that have the same side-effects as some already explored path. The main idea in RWset is to collect information of reads and writes of different variables in the program under test. If some variable $x$ is not used after some specific point in execution and the same program state excluding the information about $x$ is reached by two different execution paths, it is safe to prune one of the paths as the only difference can be due to the value of $x$ and it is not used in following execution.

Symbolic execution is also slow as seen in the case studies. Although our tool is not optimised for the fastest possible execution it can be noted that running code without instrumentation is many times faster in most cases. Luckily this slowdown can be seen as a constant factor and so it is not as severe problem as the path explosion.

On a positive side errors that require specific input values that are hard to generate by random testing can be found efficiently. However, it should be noted that when not all execution paths can be tested, our tools ability to find errors depends on random values as it blindly explores different execution paths. To improve the situation search strategies that can guide the search towards possible errors are planned to be investigated in the future.

# Chapter 7

# Conclusions

Dynamic symbolic execution is a promising approach for generating test inputs that will exercise different execution paths of a given program. In this work an instrumentation process has been developed that allows a program to be executed both concretely and symbolically at the same time. The discussed approach is based on collecting constraints for input values during execution and then using a constraint solver to obtain concrete input values if the conjunction of the constraints is satisfiable. This means that our ability to do symbolic execution relies heavily on the ability of the used constraint solver to solve the given constraints. This also limits our ability to do full symbolic execution to those cases where it is possible to expresses the collected constraints in a theory (*e.g.*, linear integer arithmetic and fixed size bit-vectors) supported by the constraint solver.

The main contributions of this work are the introduction of an alternative way to initialise symbolic objects based on lazy initialisation in comparison to the method described in [27] and a framework that allows the test generation process to be distributed to many computating nodes so that the test executions can be run concurrently. It is also described in this work how the lazy initialisation process can help in avoiding some cases where the same behaviour of a program is tested multiple times.

Many of the discussed methods have been implemented in a prototype test generation tool. From experiments it can be concluded that in its basic form dynamic symbolic execution does not currently scale well for large programs. The reason for this is that large programs commonly have so many execution paths (and without limiting the execution depth, even infinitely many of them) that it is infeasible to test them all within a reasonable time limit. With better search heuristics that decide which execution path to explore next and by utilising more information from the structure of the program under test we hope to improve our tool to achieve better scalability.

## 7.1 Future work

As future work it is planned to do research on methods for making our tool to be able to handle a broader class of programs and to improve the scalability issues. For additional support we are especially interested in extending the dynamic symbolic execution to concurrent programs. It is possible, for example, to sequentialise a concurrent program by using interleaving semantics and take advantage of well known partial-order reductions [9]. This increases the number of different execution paths to explore and combined with the path explosion problem makes obtaining good scalability even more challenging.

To improve the tools ability to do symbolic execution, adding support for bit-vectors has been planned in order to avoid the need to approximate non-linear path constraints. The accuracy of symbolic execution could also be improved by collecting completely accurate path constraints instead of having the non-aliasing assumption as discussed in Chapter 3.

To alleviate the path explosion problem, new ways are investigated to improve the developed search heuristics. Currently our tool can aim to achieve first a good branch coverage before exploring other execution paths as discussed in Chapter 4. Other kind of metrics could also be used to guide the selection of which execution path is to be tested next. By taking the structure or a formal specification of a system under test into account, it is hoped that the search process can be guided so that errors can be found faster and the user can obtain better confidence to the system even when not all the execution paths have been tested. Also doing dynamic symbolic execution compositionally as discussed in Chapter 6 seems a promising way to increase the size of programs that can be handled by our tool. Ways to prune paths from symbolic execution trees that do not increase our coverage of the behaviour a program has would also improve scalability as shown in [6].

Another direction for future work is to combine dynamic symbolic execution with runtime monitoring. In runtime monitoring a program is tested against a formal specification during a test run. To use runtime monitoring and dynamic symbolic execution efficiently together different ways are planned to be investigated to take the specification into account when the order in which the execution paths are tested is selected. This way it is hoped that the testing process can be guided to obtain a good coverage of a desired part of the specification.

# Bibliography

[1] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 367–381. Springer, 2008.

[2] Saswat Anand, Alessandro Orso, and Mary Jean Harrold. Type-dependence analysis and program transformation for symbolic execution. In *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*, pages 117–133. Springer, 2007.

[3] Saswat Anand, Corina S. Pasareanu, and Willem Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*, pages 134–138. Springer, 2007.

[4] Thomas Ball. The concept of dynamic analysis. In *7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC / SIGSOFT FSE)*, volume 1687 of *Lecture Notes in Computer Science*, pages 216–234. Springer, 1999.

[5] David L. Bird and Carlos Urias Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[6] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 351–366. Springer, 2008.

[7] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a bdd package. In *Proceedings of the 27th ACM/IEEE conference on Design automation (DAC)*, pages 40–45. ACM, 1990.

[8] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS)*, pages 322–335, New York, NY, USA, 2006. ACM Press.

[9] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 2000.

[10] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[11] Bruno Dutertre and Leonardo de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification conference (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer-Verlag, 2006.

[12] Michael D. Ernst. Static and dynamic analysis: synergy and duality. In *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, page 35. ACM, 2004.

[13] Michael Factor, Assaf Schuster, and Konstantin Shagin. Instrumentation of standard libraries in object-oriented languages: the twin class hierarchy approach. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA)*, pages 288–300. ACM, 2004.

[14] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *Proceedings of the ACM Workshop on ML*, pages 12–19. ACM, 2006.

[15] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 47–54. ACM, 2007.

[16] Patrice Godefroid and Nils Klarlund. Software model checking: Searching for computations in the abstract or the concrete. In *Integrated Formal Methods, 5th International Conference, (IFM)*, volume 3771 of *Lecture Notes in Computer Science*, pages 20–32. Springer, 2005.

[17] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, 2005.

[18] CREST homepage. `http://code.google.com/p/crest/`.

[19] Java PathFinder homepage. `http://javapathfinder.sourceforge.net/`.

[20] Yamini Kannan and Koushik Sen. Universal symbolic execution and its application to likely data structure invariant generation. In *International Symposium on Software Testing and Analysis (ISSTA)*. to appear, 2008.

[21] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

[22] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[23] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE)*, pages 416–426. IEEE Computer Society, 2007.

[24] Rupak Majumdar and Koushik Sen. LATEST: Lazy dynamic test input generation. Technical report, UC Berkeley, 2007.

[25] OpenJDK project. `http://openjdk.java.net/`.

[26] Jacob Illum Rasmussen, Gerd Behrmann, and Kim Guldstrand Larsen. Complexity in simplicity: Flexible agent-based state space exploration. In *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2007.

[27] Koushik Sen. *Scalable automated methods for dynamic program analysis.* Electronic version of doctoral thesis, Department of Computer Science, University of Illinois at Urubana Champaign, 2006.

[28] Koushik Sen and Gul Agha. CUTE and jCUTE : Concolic Unit Testing and Explicit Path Model-Checking Tools. In *18th International Conference on Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006. (Tool Paper).

[29] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In *Tests and Proofs, Second International Conference (TAP)*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

[30] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, page 13. IBM, 1999.

[31] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 97–107. ACM, 2004.

[32] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

# Appendix A

# Source Code for the Case Studies

## A.1 Method 1

```java
public static boolean isSorted(IntContainer a[]) {
    for(int i = a.length - 1; i > 0; i--)
        if(a[i].value < a[i-1].value)
            return false;

    return true;
}
```

## A.2 Method 2

```java
public static void insertionSort (IntContainer [] a)
{
  int j;

  for(int p = 1; p < a.length; p++)
  {
    IntContainer tmp = a[p];

    for(j = p; j > 0 && tmp.compareTo(a[j-1]) < 0; j--)
      a[j] = a[j - 1];

    a[j] = tmp;
  }
}
```

## A.3 Method 3

```
public static void insertionSort (IntContainer [] a)
{
  int j;

  for(int p = 1; p < a.length; p++)
  {
    IntContainer tmp = a[p];

    for(j = p; j > 0 && tmp.compareTo(a[j-1]) < 0; j--)
      if(a[j].value != 7153)
        a[j] = a[j - 1];

    a[j] = tmp;
  }
}
```