TKK Reports in Information and Computer Science

Espoo 2009 TKK-ICS-R26

AUTOMATED TEST GENERATION FOR SOFTWARE COMPONENTS

Kari Kähkönen

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

TKK Reports in Information and Computer Science

Espoo 2009 TKK-ICS-R26

AUTOMATED TEST GENERATION FOR SOFTWARE COMPONENTS

Kari Kähkönen

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

Teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta
Tietojenkäsittelytieteen laitos

Distribution:
Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science
P.O.Box 5400
FI-02015 TKK
FINLAND
URL: http://ics.tkk.fi
Tel. +358 9 470 01
Fax +358 9 470 23369
E-mail: series@ics.tkk.fi

ISBN 978-952-248-280-8 (Print)
ISBN 978-952-248-281-5 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)
URL: http://lib.tkk.fi/Reports/2009/isbn9789522482815.pdf

TKK ICS
Espoo 2009

Abstract: This report presents a method developed as a part of the LIME-project (Lightweight formal Methods for distributed component-based Embedded systems) for generating test cases for software components. The main technique 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 is collected describing the input values that will force the program to follow an unexplored execution path. By solving the collected constraints new input values are obtained allowing each test run to exercise different behaviour of the program. Based on the methods developed in this work, a tool has been implemented that can be used for automated testing of sequential Java programs.

Keywords: Automated testing, dynamic symbolic execution

CONTENTS

1 INTRODUCTION

Several automated test case generation methods have been suggested over the years. One of the simplest is random testing [2], 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 and testing in general 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 [13, 6, 12] 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.

In this report we consider a variant of symbolic execution called dynamic symbolic execution [10, 16] (also known as concolic testing). In this approach the program is executed both concretely and symbolically at the same time and the collected symbolic constraints are used to guide the concrete execution. This approach can be seen as a dynamic analysis method where as the traditional symbolic execution is a form of static analysis. The benefit of combining the symbolic and concrete executions 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.

The goal of this work is to develop a test generation method for testing of sequential Java programs. It is also discussed how the method can be extended to allow the testing process to be guided by specfications written in LIME Interface Specification Language and monitored by LIME Interface Monitoring Tool [11].

2 OVERVIEW

The purpose of this section is to familiarise the reader with the used terminology and give an overview of how dynamic symbolic execution 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. For example, assignments and subroutine calls are statements. 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 π is a sequence that consists of the first n statements of π. 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.

A *control flow graph* of a program is a graph representation of the potential execution paths in the program such that the nodes in the graph represent statements of the program and edges represent the control paths between the statements.

Example 1 *Figure 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 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.*

Different measures are used in testing to indicate how well a program under test has been exercised by the performed tests. Two measures that are relevant for understanding the test generation method described in this report are *branch coverage* and *path coverage*. Branch coverage indicates how many of

```
x = input();
x = x + 5;

if (x > 0)
    y = input();
else
    y = 10;

if (x > 2)
    if (y == 2789)
        error();
```

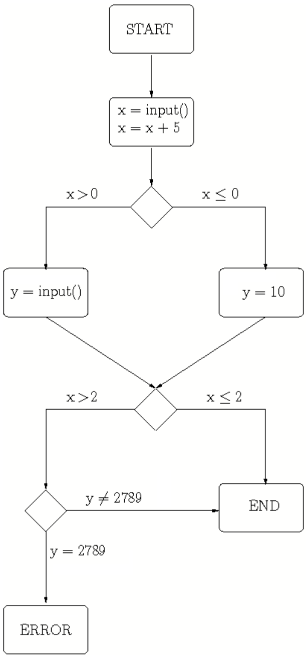


Figure 1: An example program and its control flow graph

the control structures (e.g., if-statements) in the program has evaluated both to true and false during testing. Path coverage, on the other hand, indicates if every possible control path in the code has been tested. In non-trivial programs the number of possible paths is generally too large so that full path coverage can be obtained. Furthermore, in programs with loops the number of distinct paths through the program can be infinite.

2.2 Dynamic Symbolic Execution

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 section), 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 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 the references of these objects is needed as well. For primitive data types it is enough to collect arithmetic constraints as shown with local constraints but to allow symbolic reasoning about references to input objects and their relationship to each other, it is necessary to also collect constraints that tell whether some references must or must not point to a same input object. These kind of constraints will be called *reference 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). Reference 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 the same object if and only if so required by a reference 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 reference 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 reference 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 reference 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.

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 locations 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. The path constraints are denoted in the figure by the short hand PC. If the aim is to follow an execution path that takes the true branch on the first if-statement and the false branch of the second, the corresponding path*

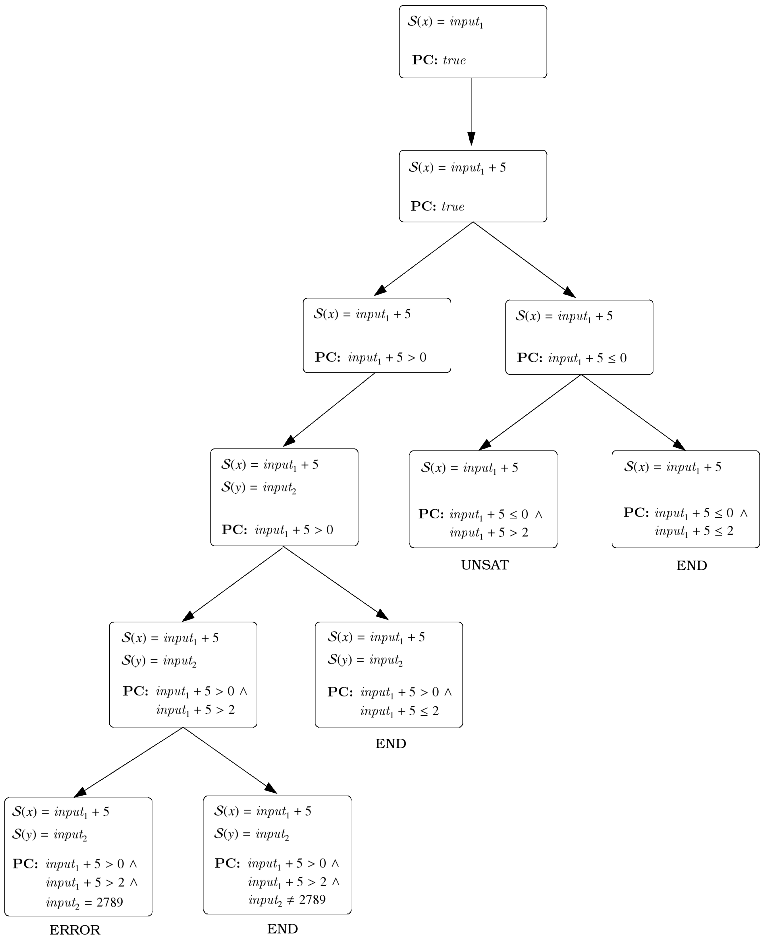

$\mathcal{S}(x) = input_1$
PC: $true$
$\mathcal{S}(x) = input_1 + 5$
PC: $true$
$\mathcal{S}(x) = input_1 + 5$
PC: $input_1 + 5 > 0$
$\mathcal{S}(x) = input_1 + 5$
PC: $input_1 + 5 \leq 0$
$\mathcal{S}(x) = input_1 + 5$
$\mathcal{S}(y) = input_2$
PC: $input_1 + 5 > 0$
$\mathcal{S}(x) = input_1 + 5$
PC: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

Figure 2: Symbolic execution tree of the first example program

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.*

2.3 Enabling Dynamic Symbolic Execution of Concrete Programs

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 section. 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 path 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

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

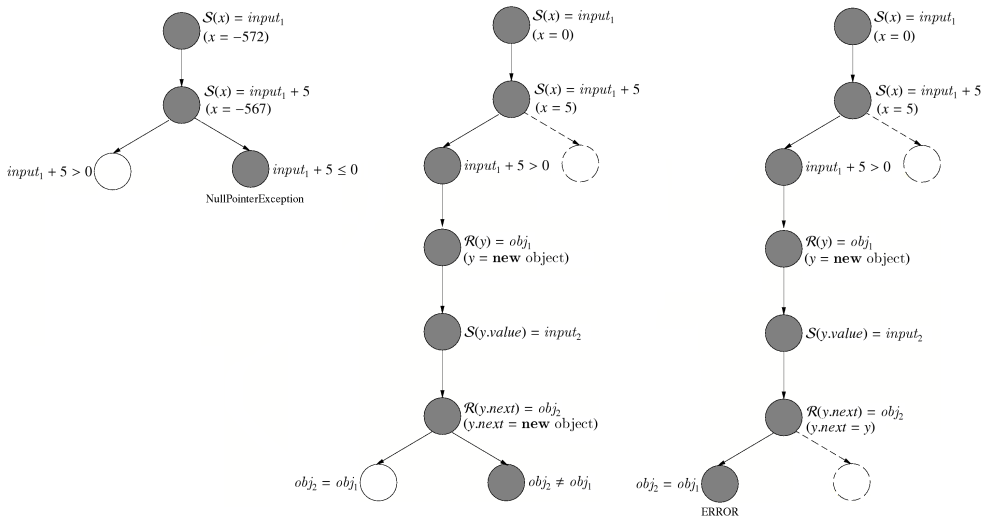
```
1 void example() {
2     int x = input();
3     SimpleList y;
4
5     x = x + 5;
6
7     if (x > 0)
8         y = input();
9     else
10        y = null;
11
12    if (y.next == y)
13        error();
14 }
```

Figure 3: Second example program

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 reference 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. During this testing process, our tool reports any uncaught exceptions as errors in the program under test.

The described test generation approach is further illustrated by an example given below. A more formal description of how both local and referece constraints are generated during execution is given in Section 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 representing a linked list as input to the system. The example code is shown in Figure 3 and the different test runs from the viewpoint of a symbolic execution tree are shown in Figure 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*



$\mathcal{S}(x) = input_1$
$(x = -572)$
$\mathcal{S}(x) = input_1 + 5$
$(x = -567)$
$input_1 + 5 > 0$
$input_1 + 5 \leq 0$
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 4: Running example of the testing algorithm

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 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 referece 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 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 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, reference 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 4. This allows the test generation algorithm to terminate.

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 section it is explained how symbolic information can stored during execution and what kind of instrumentation is needed. The discussion here considers mostly instrumenting Java programs. However, similar approach can be taken to instrument other imperative programming languages as well. 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 are made in certain cases for efficiency reasons.

The approach described in this chapter follows mostly the approach explained in [16] 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 large 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 [18] is used as this intermediate language

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	::=	< | ≤ | > | ≥| == | ! =

Figure 5: Syntax of statements in the 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 5. 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 Section 6

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` ($x == 5 \mid\mid x < 0$)) 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 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.

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 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 later of this section.

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 in the program. In other words, each primitive type local 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 local 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 [1]), 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).

In the case of object fields, a similar approach could be used as with local variables (i.e., new symbolic fields could be added to the class description to store the symbolic values). However, to minimize the amount of instrumentation needed, the symbolic memory map for object fields is implemented as a data structure that uses the object reference and field name as a key and maps the key to a symbolic value.

With input objects it is possible to have multiple references to the same object. Therefore the mapping $\mathcal{R}$ is implemented by maintaining a data structure that maps an object reference to a symbolic value, as the reference can be used as an unique identifier similar to the object field case. 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 $\mathit{input}_1 + 5$ for a variable x a symbolic value s_0 is used and the information that $s_0 = \mathit{input}_1 + 5$ 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 $\mathit{input}_1 + \mathit{input}_1 = 2 \times \mathit{input}_1$ similarly to [16, 17] 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 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 selector. The details of the test selector are given in Chapter 4.

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 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 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 exe-

Before instrumentation	**After instrumentation**
begin	$\mathcal{I}$ = receive inputs; $i = k = j = inputNumber = 0$; $\mathcal{S} = \mathcal{M} = \mathcal{R} = []$; begin
$v = e$;	EXECUTEASSIGNMENT(v, e); $v = e$;
v = input;	v = GETINPUT(v);
if v_1 *op* v_2 goto *l*;	EXECUTECONDITION(op, v_1, v_2); if v_1 *op* v_2 goto *l*;

goto;	CHECKGOTOCOUNT();
	goto;

end	REPORTEND();
	end

Table 1: Instrumentation of statements

cuted to make sure that the correct execution path prefix will be followed. The selector can alternatively send also the path constraint that can be used to compute the input values. This alternative is discussed in more detail in Section 4.5. The input values received from the test selector are denoted by an input 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.*, the system uses input values that are not received from the test selector) the test run is not guaranteed to follow the expected execution path. To 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 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 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 6.

With every assignment statement it is necessary to make sure that the symbolic values are also updated. Figure 7 shows how this is done. When a

```
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 6: Getting correct input values during execution

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 and if it is not, the assignment is executed only concretely 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 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.

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 selector as illustrated in Figure 8. The method first checks whether the if-statement operates on symbolic values or not. If only concrete values are used, the test selector needs not to be informed as the statement does not affect the symbolic execution tree of the program. If symbolic val-

```
EXECUTEASSIGNMENT$(v, e)$
match $(e)$
    case $c$: //$c$ is a constant value
        $\mathcal{S} = \mathcal{S} - \mathrm{v}$;
    case $v_1$: //$v_1$ is a variable
        if $(v_1 \in \textbf{domain}(\mathcal{S}))$
            $\mathcal{S} = \mathcal{S}[v \mapsto \mathcal{S}(v_1)]$;
        else
            $\mathcal{S} = \mathcal{S} - v$;
    case $v_1$ op $v_2$:
        if (op is not supported by the constraint solver)
            $\mathcal{S} = \mathcal{S} - v$;
        else if$(v_1 \in \textbf{domain}(\mathcal{S}) \wedge v_2 \in \textbf{domain}(\mathcal{S}))$
            $\mathcal{S}[v \mapsto s_i]$; //$s_i$ is a new symbolic value
            $i = i + 1$;
            if (op $\in \{*, /\} \wedge$ only linear constraints supported)
                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$(v_1 \in \textbf{domain}(\mathcal{S}))$
            $\mathcal{S}[v \mapsto s_i]$;
            $i = i + 1$;
            REPORT$(\mathcal{S}(v) = \mathcal{S}(v_1)$ op $v_2)$;
        else if$(v_2 \in \textbf{domain}(\mathcal{S}))$
            $\mathcal{S}[v \mapsto s_i]$;
            $i = i + 1$;
            REPORT$(\mathcal{S}(v) = v_1$ op $\mathcal{S}(v_2))$;
        else
            $\mathcal{S} = \mathcal{S} - v$;
```

Figure 7: Executing symbolic assignments

ues are used, the local constraint for the true branch is reported to the test 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 selector knows which of the branches it will keep expanding. Note that on line 2 the information of the taken branch is used in construction 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 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

```
executeCondition($op, v_1, v_2$)
branchTaken = evaluate($v_1$ op $v_2$);
constructBranchBitvector(branchTaken);
if ($v_1 \in$ domain($\mathcal{S}$) $\wedge$ $v_2 \in$ domain($\mathcal{S}$))
    report($\mathcal{S}(v_1)$ op $\mathcal{S}(v_2)$, branchTaken);
else if($v_1 \in$ domain($\mathcal{S}$))
    report($\mathcal{S}(v_1)$ op $v_2$, branchTaken);
else if($v_2 \in$ domain($\mathcal{S}$))
    report($\mathcal{S}(v_2)$ op $v_1$, branchTaken);
```

Figure 8: Executing if-statements symbolically

Before instrumentation	**After instrumentation**
o = `input`;	*o* = getSymbolicObject(*o*);

v = *o.field*;	lazyInitialize(*o*);
	executeAssignment(*v*, *o.field*);
	v = *o.field*;

o.field = *e*;	lazyInitialize(*o*);
	executeAssignment(*o.field*, *e*);
	o.field = *e*;

if o_1 *op* o_2 *goto l;*	executeObjectCondition(*"op"*, o_1, o_2);
	if o_1 *op* o_2 *goto l;*

Table 2: Instrumentation of object statements

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.

Symbolic Execution with Objects

The additional instrumentation needed for symbolic execution with input objects is shown in Table 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 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 selector can simply give a concrete value that is assigned

```
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 9: Getting symbolic objects as inputs

to a variable but with object inputs no such concrete value can be given. In place of concrete values the test 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 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 9 shows the algorithm for creating a new input object. Similarly to the primitive input case it is first checked if the test 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 is given to the object. This is because all input objects are assumed to be distinct unless required otherwise by the test 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 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 reference constraints the EXECUTEOBJECTCONDITION method presented in Figure 10 is executed before any if-statement that compares objects references instead of primitive values. Our tool collects reference constraints that can be only of the form $o_1 = o_2$, $o_1 \neq o_2$, $o_1 = \mathit{null}$ and $o_1 \neq \mathit{null}$, 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 reference constraint based on the symbolic values of the objects. As null object references do not have symbolic values associated to them, they are

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

Figure 10: Generating reference constraints

handled as a special case (in lines 6 and 8). The generated reference constraint and the branch that the concrete test run will take are then reported to the test selector.

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 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 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.

Before instrumentation	After instrumentation
$v = method(v_1, ..., v_n);$	$\textsc{CheckInvokeCount}();$ $push(\mathcal{S}(v_1)); ...; push(\mathcal{S}(v_n));$ $v = method(v_1, ..., v_n);$ $\mathcal{S}[v \mapsto pop()];$
$method(v_1, ..., v_n)$ $\{$ ... **return** v; $\}$	$method(v_1, ..., v_n)$ $\{$ $\mathcal{S}[v_n \mapsto pop()]; ...; \mathcal{S}[v_1 \mapsto pop()];$ ... $push(\mathcal{S}(v));$ **return** v; $\}$

Table 3: Instrumentation of methods and method calls

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 reference constraints are only constructed at branching points of a program similarly to [16] where a program code, a[i] = 0; a[j] = 1; if (a[i] == 0) 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 reference 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 [5], 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.

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.

3.4 Objects with Invariants

Generating object inputs that must satisfy an 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 cause 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 [19] 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 reference 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.

4 GENERATING INPUTS

In this chapter the focus is moved to describing the second main part of our tool, the test 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 [16, 10], 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 [17]. Constructing a symbolic execution tree allows the test 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 to communicate with the test selector and not with each other. This will be further discussed in this chapter and in Section 6.

The general working principle of the test selector is shown in Figure 11.

```
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     mark $n$ visited;
```

Figure 11: General testing algorithm

The test 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 4.3. After an unvisited node is selected, a path constraint corresponding to it is formed by collecting local and reference 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 events observed during the test run.

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 guaratentee that the same execution path is not tested multiple times, and

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.
Depth limit	The maximum execution depth has been reached.
Error	The Java program under test has terminated due to an uncaught exception.
End	The program has terminated normally.

Table 4: Message types and their descriptions

- it can be used to check that a test run follows the execution path predicted by the branches in the symbolic execution tree.

The third characteristic is needed 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 section.

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. Futhermore, the search strategies use different auxilary data structures to efficiently select unvisited nodes from the binary tree.

4.2 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 selector can receive are shown in Table 4 as a summary from Chapter 3. Assume that a node has been 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 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.

Depth limit and error messages denote that the current path is not ex-

```
1  List l1 = input();
2  List l2 = input();
3  int x   = input();
4
5  if (x == 5) {
6    if (l1 != null)
7      if (l2 != null) {
8        l1.value = l2.value
9
10         if (l1 == l2)
11           print(l1.value);
12   }
13 }
```

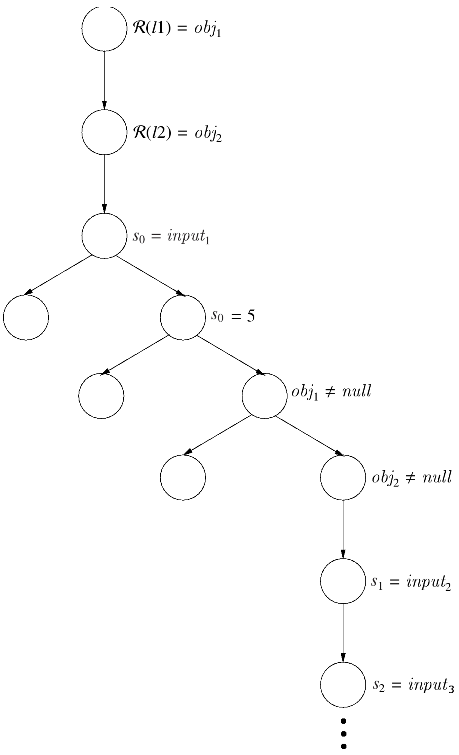


Figure 12: Symbolic object example

plored 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 12. Let us assume that the test 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 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 solve this problem, a new branching point is added to the symbolic execution tree when the initialization of a new input object with symbolic value obj_i is reported to have happened. This special branch point will be refered to as *initialization point* of obj_i. The initialization points have two child nodes. The first child corresponds to the execution path where the initialization does happen and it is given a reference constraint that fixes the logical address of the input object to an unique value (e.g., $obj_2 = 2$). The other child node is given a negation of the reference constraint and the subtree starting from this node is intended to contain all the symbolic execution paths where the input object at hand is the same as some other input object created earlier during the execution. However, this second branch is not marked as unvisited when a new initialization point is created as it is not known to which input objects the current object reference might point to.

Example 6 *Let us consider the following code snippet:*

```
1: List l1 = input();
2: List l1 = input();

3: l1.value = l2.value;

4: if (l1 == l2)
5:   ...
6: else
7:   ...
```

The program takes two List objects as input and uses them on line 3 causing the lazy initialization of both input objects to occur. After executing line 3 and line 4 the symbolic execution tree of the program will contain two branches as shown in Figure 13. The initialization points are marked with gray colour in the picture.

As can be seen from the previous example, the fixed values assigned at initialization cause that the path constraint $obj_1 = 1 \land obj_2 = 2 \land obj_1 = obj_2$ is unsatisfiable. However, it is possible to follow the true branch at line 4 of the example if the two input objects are the same. Therefore, whenever a path constraint is found to unsatifiable due to a conflict caused by a reference constraint of the form $obj_i = obj_j$, the symbolic execution tree is searched for the initialization points of the two objects named in the constraint. The initialization point located deeper in the symbolic execution tree is then extended with a new branch to a node that contains the constraint $obj_i = obj_j$.

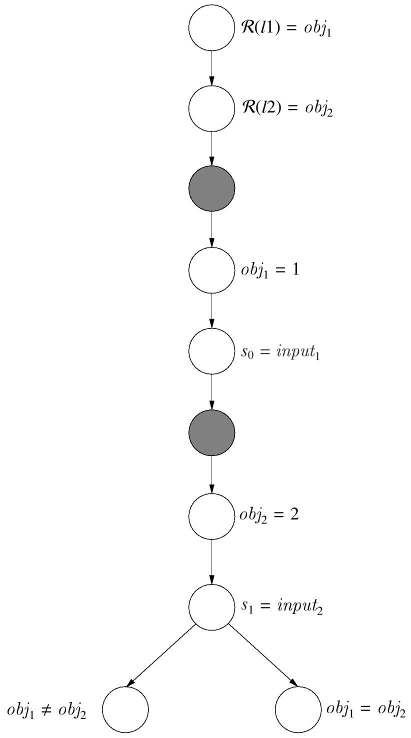


Figure 13: Symbolic execution tree and input objects

If the path constraint corresponding to this new node is satisfiable, the execution paths where the two objects are equal are explored. Figure 14 shows the symbolic execution tree of the previous example that contains satisfiable symbolic execution paths for both outgoing branches of the if-statement at line 4.

It is also possible to create the initialization points when the input objects are created. For example, at lines 1 and 2 in the example above. The approach used in jCUTE can be seen to fit in this category, although the construction of symbolic execution trees there is different due to not using lazy initialization and limiting the search strategies to depth-first search. It is however, beneficial to postpone the branching caused by the initialization points to the point where the object are used for the first time. To see this, let us consider the example program in Figure 12. If the false branch of the `if (x == 5)` statement at line 5 would contain a large portion of code that does not use the input objects $l1$ and $l2$, creating branches at lines 1 and 2 would cause this portion of code to explored multiple times with the same symbolic values. By postponing the branching after the if-statement at line 5, the subtree corresponding to the code behind the false branch is explored only once.

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 event 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

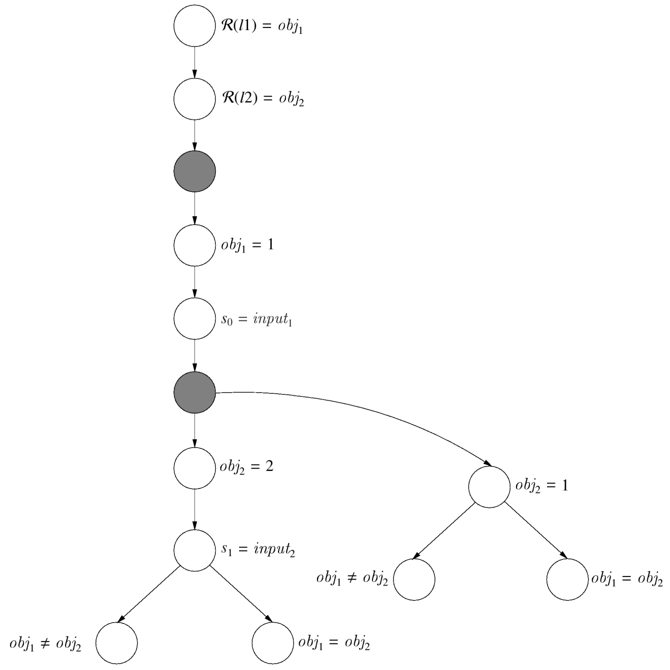


Figure 14: Symbolic execution tree and input objects

```
 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 15: Example of execution failing to follow the correct path

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 15. 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 $y > x \wedge x > 10$ 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 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 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.

4.3 Search Strategies

As each test run can potentially add multiple new branches that can be explored to the symbolic execution tree, the test 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.

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 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

[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.

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.

Selecting Unvisited Nodes Based on Priorities

The third search strategy in our tool 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 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.

The priorities for the unvisited branches can be given randomly or based on some heuristic that tries to guide the testing process. An example of such a heuristic is discussed in Section 5. If priorities are assigned randomly it has a downside 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.

Selecting Unvisited Nodes Randomly
To address the problems with the search strategy based on random priorities, we have also implemented a search that simply selects one of the unvisited branches randomly.

4.4 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.

4.5 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.

Local and Reference 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 referece 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 reference 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 a reference 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 reference 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 a reference 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 reference 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.

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 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 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.

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* [16] 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. If it 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 reference 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 the 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 [17] additional simplification approaches based on using BDD [3] representations of logical connectives and hash-consing [9] among others as discussed.

In CUTE [16] 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 [16], 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 [16] that the constraints given to the constraint solver were reduced on average to one-eight the size of the original path constraint in many cases.

5 TEST GENERATION FOR PROGRAMS WITH SPECIFICATIONS

The testing method described in the earlier sections reports uncaught exceptions as errors (i.e., it generates tests to see if the program can crash). Dynamic symbolic execution can be greatly enhanced if it is combined with runtime monitoring to check if given specifications hold during the test runs. In the LIME project a specification language has been developed together with a runtime monitoring tool [11] that allows the user to use propositional linear temporal logic (PLTL) and regular expressions to specify both external usage and internal behavior of a software component.

The LIME interface specification language allows the user to write specifications that are not complete models of the system and to target the specifications to those parts of the systems that are seen as important to be tested. This means that only a portion of the execution paths in the program may cause the specifications to be monitored. Also, the specifications provide additional information about the system that could be used to indicate when the program is close to a state that violates the specifications. Therefore we have extended the dynamic symbolic execution method to take LIME interface specifications into account so that the testing can be guided towards those execution paths that cause specifications to be monitored and especially towards those paths that can potentially cause the specifications to be violated.

To guide the testing, the LIME Interface Monitoring Tool (LIMT) [14] was extended to provide a method to compute a heuristic value to indicate how close the current execution is to violating the monitored specifications. The details of computing the heuristic values is given in [15]. On dynamic symbolic execution side the instrumentation described in Section 2 is augmented with a call to LIMT to obtain the heuristic value at every branching statement in the execution where symbolic values are used. The test selector is then notified about the heuristic value and it can use the value to assign a priority to the unvisited branch resulting from executing the branching statement. In otherwords, the heuristic value is used as the priority for unvisited branches in the symbolic execution tree. By using the priority based search strategy described in Section 4, those execution paths are explored first that have the highest probability to lead to a state that violates the specifications according to the heuristic used to compute the priority values.

The described approach to guide the testing process has the following weakness. It is likely that when the top most branch in the symbolic execution tree is created, the execution is still far from a state that can potentially violate a specification. Therefore the top most branch that was not followed by the first concrete test run is likely to get a low priority value. This can cause the search strategy to ignore the second half of the symbolic execution tree until the first one has been explored. To address this problem, the test selector can use a combination of the priority based search and random search to select the next branch to be explored. For a user specified fraction of times, the next branch to be explored is selected randomly instead of based on the priority values. This way the problems of greedily choosing the most promising executions paths is alleviated at least partly.

6 IMPLEMENTATION

In this chapter the implementation details of our tool called LIME Concolic Tester (LCT) that is based on the methods described in Chapters 3 and 4 are discussed. The limitations in the tool that the users should be aware of are also discussed.

6.1 Structure of the Testing System

The structure of LCT is shown in Figure 16 and it can be seen as consisting three main parts: the instrumenter, the test selector and the test executors. The instrumenter is based on a tool called Soot [18], 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 = LCT.getInteger()` is used to get an int type input value for a variable x, and

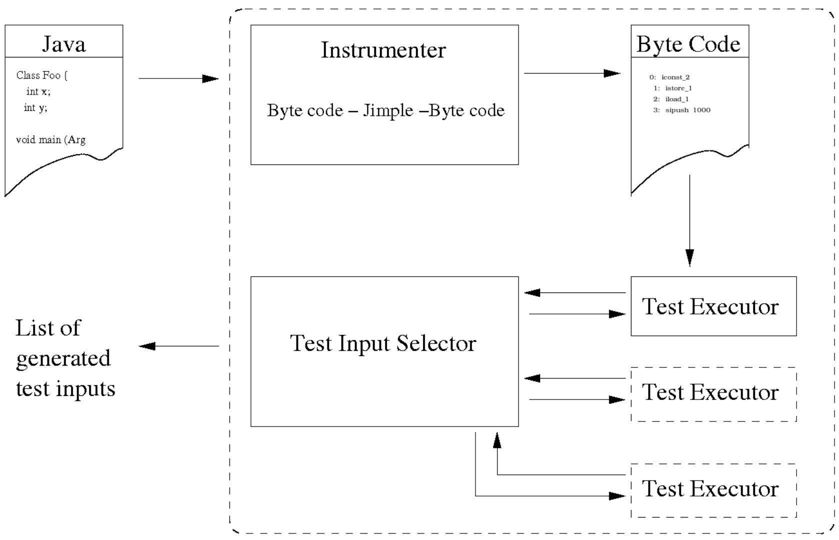


Figure 16: Structure of the testing system

- `List l = LCT.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 as most constraint solvers do not provide native support for floating point variables.

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 selector to report the collected constraints and to receive 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 either Yices [7] or Boolector [4] as a constraint solver and 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.

6.2 Generating JUnit Test Cases

The tool also provides the possibility to generate JUnit tests based on the input values generated during dynamic symbolic execution. The support for JUnit tests is limited to unit testing methods that take argument values that can be generated by LCT. To generate JUnit tests for a selected method, the tool first creates automatically a test driver that calls the method with input values computed by LCT. The generated input values are then stored and used to generate JUnit tests that can be executed even without LCT.

6.3 Computing Branch Coverage

The tool computes also branch coverage obtained during the testing. This implemented by assigning a unique static identifier to each of the branching statements in the program during the instrumentation. It is then recorded whether all the true and false branches of these statements have been exercised. As the branching statements are identified during instrumentation, the branch coverage reported describes the coverage over all the instrumented parts of the program. If the program uses large libraries, the branch coverage of these libraries is included in the result. Also if LIME interface specfications are monitored using LIMT, the monitoring does some instrumentation to the program code as well and the the branch coverage of this modified version is displayed. However, the tool identifies some of the code inserted by LIMT and ignores them in the branch coverage computation. This identification is not fully accurate and therefore the branch coverage should be seen as approximate when LIME interface specifications are used.

6.4 Limitations

LCT has currently some limitations regarding the use of core classes in Java (e.g., classes in java.lang package) and by default it does not instrument them. The reason for not instrumenting core classes is that in many Java Virtual Machines (JVM) they cannot be modified freely as most Java Virtual Machines are very sensitive to modifications to the core system classes (e.g., the load order of classes during bootstrapping may change due to instrumentation) and this can cause the JVM to crash. Furthermore, instrumenting the core classes means that the instrumented code that also uses core classes would use their rewritten versions which can cause complications.

To address this limitation, we have implemented custom versions of Integer, Long, Byte, Boolean and Short classes that can be instrumented freely.

The program under test is then modified to use the custom versions of these classes instead of the original counterparts. This approach can be seen as a lightweight counterparts of twin class hierarchy approach presented in [8].

Currently instrumenting other classes is not supported but new custom implementations of different classes can be added in the future. If the system under test uses classes that are not instrumented, they are executed only concretely. This prevenets tracking symbolic values of variables inside the uninstrumented code and can cause some execution paths to be left untested. The core class replacement approach used in LCT introduces one additional limitation. If the system under test contains uninstrumented code that expects one of the core class instances that have been replaced to be received from the instrumented code, the type of the instances is not the same (i.e., the replamecement of Integer class is LCTInteger) and this can cause the program under test to crash. Therefore, it is recommended that the core class replacement is used only with programs that can be fully instrumented or where it can be guaranteed that the replacement approach does not cause illegal type convertions.

7 CONCLUSIONS

Dynamic symbolic execution is a promising approach for generating test inputs that will exercise distinct execution paths of a given program. In this report an instrumentation process has been developed that allows a program to be executed both concretely and symbolically at the same time. It is also described how the symbolic execution can be used to generate test inputs to a system under test by utilizing off-the-shelf constraint solvers. The described method can also be used in conjunction with specifications written with LIME interface specification language and monitored by LIME Interface Monitoring Tool.

–

REFERENCES

[1] 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)*, 2007.

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

[3] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a bdd package. In *DAC*, pages 40–45, 1990.

[4] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In Stefan Kowalewski and Anna Philippou,

editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer, 2009.

[5] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM Press.

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

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

[8] Michael Factor, Assaf Schuster, and Konstantin Shagin. Instrumentation of standard libraries in object-oriented languages: the twin class hierarchy approach. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA*, pages 288–300. ACM, 2004.

[9] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In Andrew Kennedy and François Pottier, editors, *ML*, pages 12–19. ACM, 2006.

[10] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.

[11] Kari Kähkönen, Jani Lampinen, Keijo Heljanko, and Ilkka Niemelä. The LIME Interface Specification Language and Runtime Monitoring Tool. In *Proceedings of the 9th Internatinal Workshop on Runtime Verification*, volume 5779 of *Lecture Notes in Computer Science*, pages 93–100, 2009.

[12] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

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

[14] Jani Lampinen, Sami Liedes, Janne Kauttio, Kari Kähkönen, and Keijo Heljanko. Incremental specification of software components and interfaces. Technical Report TKK-ICS-R25, Helsinki University of Technology, 2009.

[15] Olli Saarikivi. Design and implementation of a heuristic for directing dynamic symbolic execution, 2009. LIME project deliverable.

[16] Koushik Sen. *Scalable automated methods for dynamic program analysis*. Electronic version of doctoral thesis, University of Illinois, 2006.

[17] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In Bernhard Beckert and Reiner Hähnle, editors, *TAP*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

[18] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In Stephen A. MacKay and J. Howard Johnson, editors, *CASCON*, page 13. IBM, 1999.

[19] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *ISSTA*, pages 97–107, 2004.

TKK REPORTS IN INFORMATION AND COMPUTER SCIENCE

TKK-ICS-R16 Antti E. J. Hyvärinen
Approaches to Grid-Based SAT Solving. June 2009.

TKK-ICS-R17 Tuomas Launiainen
Model checking PSL safety properties. August 2009.

TKK-ICS-R18 Roland Kindermann
Testing a Java Card applet using the LIME Interface Test Bench: A case study.
September 2009.

TKK-ICS-R19 Kalle J. Palomäki, Ulpu Remes, Mikko Kurimo (Eds.)
Studies on Noise Robust Automatic Speech Recognition. September 2009.

TKK-ICS-R20 Kristian Nybo, Juuso Parkkinen, Samuel Kaski
Graph Visualization With Latent Variable Models. September 2009.

TKK-ICS-R21 Sami Hanhijärvi, Kai Puolamäki, Gemma C. Garriga
Multiple Hypothesis Testing in Pattern Discovery. November 2009.

TKK-ICS-R22 Antti E. J. Hyvärinen, Tommi Junttila, Ilkka Niemelä
Partitioning Search Spaces of a Randomized Search. November 2009.

TKK-ICS-R23 Matti Pöllä, Timo Honkela, Teuvo Kohonen
Bibliography of Self-Organizing Map (SOM) Papers: 2002-2005 Addendum.
December 2009.

TKK-ICS-R24 Timo Honkela, Nina Janasik, Krista Lagus, Tiina Lindh-Knuutila, Mika Pantzar, Juha Raitio
Modeling communities of experts. December 2009.

TKK-ICS-R25 Jani Lampinen, Sami Liedes, Kari Kähkönen, Janne Kauttio, Keijo Heljanko
Interface Specification Methods for Software Components. December 2009.

ISBN 978-952-248-280-8 (Print)
ISBN 978-952-248-281-5 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)