Symbolic & Concolic Execution

Symbolsk og Concolic eksekvering

Bachelor report (15 ECTS) in Computer Science
Department of Computer Science, Aarhus University

Søren Baadsgaard · 201305284
Advisor: Magnus Madsen

June 15, 2019

Abstract

Symbolic execution is a technique for systematically exploring all execution paths of a program, and for each path generate concrete input values that will execute along this path. This serves as a strong tool for testing programs, as it is often difficult to come up with a proper selection of testing inputs that cover the entire program. We describe the theory behind symbolic execution and discuss a number of limitations to the technique, such as the path-explosion problem, and the ability to decide which paths are feasible. Furthermore we describe a different technique called concolic execution, that combines concrete and symbolic execution to mitigate some of the challenges of pure symbolic execution. Finally, we demonstrate symbolic execution by describing and comparing an implementation of both a concrete and symbolic interpreter for a small toy language.

Contents

Chapter 1

Introduction

Proper testing of programs requires a good selection of input values that cover all edge cases of the program to ensure that it behaves as expected. Choosing such a selection is often a difficult and time consuming task, and so we risk ending up with a lackluster choice of test cases, which leaves large parts of the program untested. The goal of this report is to study *symbolic execution*, which is a classical technique to systematically explore all execution paths of a program and generate concrete input values that will follow these execution paths and thereby obtain test cases that properly cover the entire program. To this, the program is executed symbolically by replacing the input values with symbolic values that represents arbitrary concrete values. Whenever a conditional statement is executed, there are two potential execution paths to follow. If both paths are feasible, the execution splits into two executions, each following a different path. Each execution keeps track of the constraints that the input values must satisfy to follow that path, and from these constraints we can derive concrete input values that will execute along the same path.
We will also study a modern approach called *concolic execution* that combines concrete and symbolic execution. In this technique, the program is executed with initially random concrete input values. During the execution, we maintain both symbolic and concrete values for all variables, and whenever the execution branches, the choice of branch is registered in a list of constraints, using the symbolic values. At the end of the execution, this list of constraints is used to generate a new set of concrete input values that will cause the program to execute along a different path. This process is repeated until all execution paths have been explored.

Chapter 2

Motivation

In this chapter we will consider a motivating example that illustrates the usefulness of symbolic execution as a software testing technique.

2.1 A motivating example

Consider a company that sells a product at a unit price 2. If the total price of an order is greater than or equal 16, a discount of 10 is applied. However, the total price including the discount may not be lower than some specified minimum, that may change from order to order. The following program ComputeTotal takes integer inputs *units* and *minumum*, and computes the total price based on this pricing scheme.

```
1: procedure ComputeTotal(units, minimum)
2:     total := 2 · units
3:     if total ≥ 16 then
4:         total := total − 10
5:         assert total ≥ minimum
6:     end if
7:     return total
8: end procedure
```

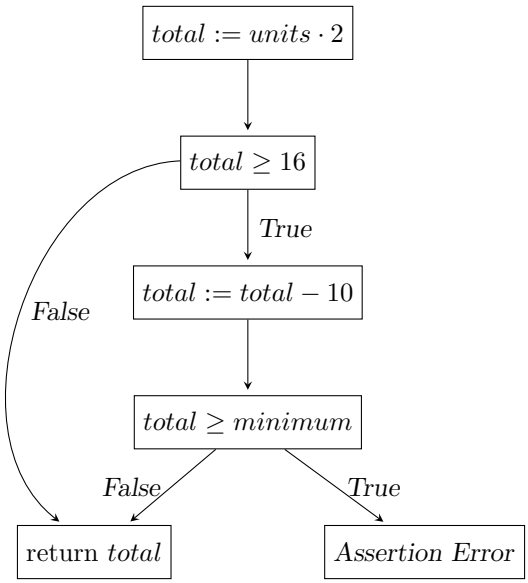

Figure 2.1: Control-flow graph for COMPUTETOTAL

We would like to know if this program ever fails due an assertion error, so we have to figure out if there exist integer inputs for which the program reaches the *Assertion Error* node in the control-flow graph shown in figure 2.1. We might try to run the program on different input values, e.g $(units = 8, minimum = 5)$, $(units = 7, minimum = 10)$. These input values does not cause the program to fail, but we are still not convinced that it wont fail for some other input values. By studying the program for some time, we realize that if the input satisfies the following constraints

$$2 \cdot units \geq 16$$
$$2 \cdot units - 10 < minimum$$

it will fail. This is the case e.g for $(units = 8, minimum = 7)$. This realization was not immediately obvious, and for more complex programs, answering the same question is even more difficult. The key insight is that the conditional statements dictates which execution path the program will follow. In this report we will present *symbolic execution*, which is a technique to systematically explore different execution paths and generate concrete input values that will follow these same paths.

Chapter 3

Principles of symbolic execution

In this chapter we will cover the theory behind symbolic execution. We will start by describing what it means to *symbolically* execute a program and how we deal branching. We will also explain the connection between a symbolic execution of a program, and a concrete execution. We shall restrict our focus to programs that takes integer values as input and allows us to do arithmetic operations on such values. In the end we will cover the challenges and limitations of symbolic execution that arises when these restrictions are lifted.

3.1 Symbolic executing of a program

During a normal execution of a program, input values consists of integers. During a symbolic execution we replace concrete values by symbols e.g α and β, that acts as placeholders for actual integers. We will refer to symbols and arithmetic expressions over these as *symbolic values* (Cadar and Sen, 2013) (King, 1976). To illustrate this, we consider the following program that takes parameters a, b and c and computes their sum. The computation may seem unnecessarily complicated, but we do it this way to clearly what we mean by symbolic values, and how they relate to concrete values.

procedure ComputeSum(a, b, c)
    $x := a + b$
    $y := b + c$
    $z := x + y - b$
    return z
end procedure

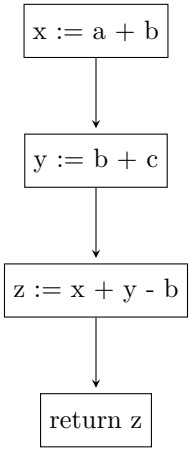

Figure 3.1: Control-flow graph for ComputeSum

Lets consider running the program with concrete values $a = 2, b = 3$ and $c = 4$. We then get the following execution: First we assign $a + b = 5$ to the variable x. Then we assign $b + c = 7$ to the variable y. Next we assign $x + y - b = 9$ to variable z and finally we return $z = 9$, which is indeed the sum of 2, 3 and 4.
Let us now run the program with symbolic input values α, β and γ for a, b and c respectively.

We then get the following execution: First we assign $\alpha + \beta$ to x. We then assign $\beta + \gamma$ to y. Next we assign $(\alpha + \beta) + (\beta + \gamma) - \beta$ to z.Finally we return $z = \alpha + \beta + \gamma$. We can conclude that the program correctly computes the sum of a, b and c, for any possible value of these.

3.2 Execution paths and path constraints

The program that we considered in the previous section contains no conditional statements, which means it only has a single possible execution path. In general, a program with conditional statements $s_1, s_2, \ldots, s_n$ with conditions $q_1, q_2, \ldots, q_n$, will have several execution paths that are uniquely determined by the value of these conditions. In symbolic execution, we model this by introducing a *path-constraint* for each execution path. The *path-constraint* is a list of boolean expressions $[q_1, q_2, \ldots, q_k]$ over the symbolic values, corresponding to conditions from the conditional statements along the path. At the start of an execution, the *path-constraint* is simply the empty list, since we have not encountered any conditional statements. to continue execution along a path, $q_1 \wedge \ldots \wedge q_k$ must be *satisfiable*. To be *satisfiable*, there must exist an assignment of integers to the symbols, such that the conjunction of the conditions evaluates to true. For example, $q = (2 \cdot \alpha > \beta) \wedge (\alpha < \beta)$ is satisfiable, because

we can choose $\alpha = 10$ and $\beta = 15$ in which case q evaluates to *true*. On the other hand $q' = (2 \cdot \alpha < 4) \wedge (\alpha > 4)$ is clearly not satisfiable since the first condition stipulates that $\alpha < 2$ and the second condition stipulates that $\alpha > 4$.

Whenever we reach a conditional statement with condition q_k, we consider the two following expressions:

1. $q_1 \wedge q_2 \wedge \ldots \wedge q_k$
2. $q_2 \wedge q_2 \wedge \ldots \neg q_k$

This gives a number of possible scenarios:

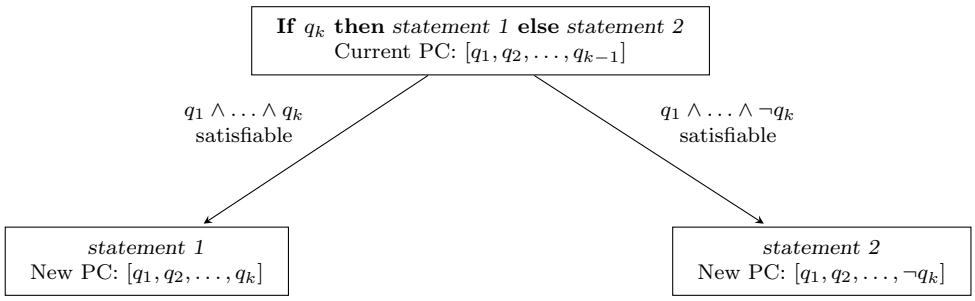

Figure 3.2: Abstract overview of the symbolic execution of an *if*-statement, which potentially leads to two new execution paths, each with a new *path-constraint* .

- **Only the first expression is satisfiable**: Execution continues with a new *path-constraint* $[q_1, q_2, \ldots, q_k]$, along the path corresponding to q_k evaluating to to *true*.
- **Only the second expression is satisfiable**: Execution continues with a new *path-constraint* $[q_1, q_2, \ldots, \neg q_k]$, along the path corresponding to q_k evaluating to to *false*.
- **Both expressions are satisfiable**: In this case, the execution can continue along two paths; one corresponding to the condition being *false* and one being *true*. At this point we *fork* the execution by considering two different executions of the remaining part of the program. Both executions start with the same environment and *path-constraints* that are equal up to the final condition. One will have q_k as the final condition and the other will have $\neg q_k$. These two executions will continue along different execution paths that differ from this conditional statement and onward (King, 1976).

To illustrate this, we consider the program from the motivating example, that takes input parameters *units* and *minimum*:

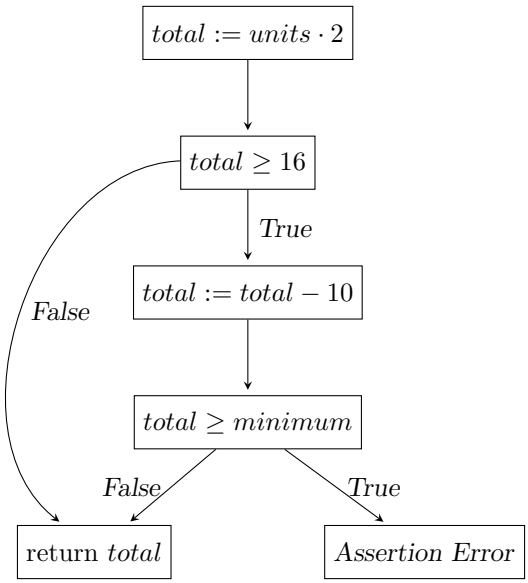

Figure 3.3: Control-flow graph for COMPUTETOTAL

We assign symbolic values α and β to *units* and *minimum* respectively, and get the following symbolic execution:

First we assign $2 \cdot \alpha$ to *total*. We then reach an *if*-statement with condition $\alpha \cdot 2 \geq 16$. To proceed, we need to check the satisfiability of the following two expressions:

1. $(\alpha \cdot 2 \geq 16)$
2. $\neg(\alpha \cdot 2 \geq 16)$.

Since both these expressions are satisfiable, we need to fork. We continue execution with a new *path-constraint* $[(\alpha \cdot 2 \geq 16)]$, along the path corresponding to the condition evaluating to *true*. We also start a new execution with the same environment and a *path-constraint* equal to $[\neg(\alpha \cdot 2 \geq 16)]$. This execution will continue along the path corresponding to the condition evaluating to *false*, and it immediately reaches the return statement and returns $\alpha \cdot 2$. The first execution assigns $2 \cdot \alpha - 10$ to *total* and then reach an *assert*-statement with condition $2 \cdot \alpha - 10 \geq \beta$. We consider the following expressions:

1. $(\alpha \cdot 2 \geq 16) \wedge (((2 \cdot \alpha) - 10) \geq \beta)$
2. $(\alpha \cdot 2 \geq 16) \wedge \neg(((2 \cdot \alpha) - 10) \geq \beta)$

Both of these expressions are satisfiable, so we fork again. In the end we have discovered all three possible execution paths with the following *path-constraints*:

1. $\neg(\alpha \cdot 2 \geq 16)$
2. $(\alpha \cdot 2 \geq 16) \wedge ((2 \cdot \alpha - 10) \geq \beta)$
3. $(\alpha \cdot 2 \geq 16) \wedge \neg((2 \cdot \alpha - 10) \geq \beta)$.

The first two *path-constraints* corresponds to the two different paths that leads to the return statement, where the first one returns $2 \cdot \alpha$ and the second one returns $2 \cdot \alpha - 10$. Inputs that satisfy these, do not result in a crash. The final *path-constraint* corresponds to the path that leads to the *Assertion Error*, so we can conclude that all input values that satisfy these constraints, will result in a program crash.
We can now solve each of these path-constraints to obtain concrete input values that will execute along the given path. Consider for example the second path-constraint. The first condition tells us that $\alpha \geq 8$, so we can choose $\alpha = 8$. From the second condition we then get that $6 \geq \beta$, so we can choose $\beta = 6$. A concrete execution of the program with $units = 8$ and $minimum = 6$ will then follow the execution path corresponding to the second path-constraint. We can do the same for the remaining two path-constraints, and ind the end we will have a set of concrete input values for each possible execution path.

3.3 Limitations of symbolic execution

So far we have only considered symbolic execution of programs with a small number of execution paths. Furthermore, the constraints placed on the input symbols have all been linear expressions. In this section we will cover the challenges that arise when we consider more general programs.

3.3.1 The path explosion problem

Since each conditional statement in a given program can result in two different execution paths, the total number of paths to be explored is potentially exponential in the number of conditional statements. For this reason, the running time of the symbolic execution quickly gets out of hands if we explore all paths. The challenge gets even greater if the program contains a looping statement. In this case, the number of execution paths is potentially infinite (Cadar and Sen, 2013). We illustrate this by considering the following program that computes a^b for integers a and b, with symbolic values α and β for a and b:

This program contains a *while*-statement with condition $i \leq b$. The $k'th$ time we reach this statement we will consider the following two expressions:

1. $(1 \leq \beta) \wedge (2 \leq \beta) \wedge \ldots \wedge (k - 1 \leq \beta)$
2. $(1 \leq \beta) \wedge (1 \leq \beta) \wedge \ldots \wedge \neg(k - 1 \leq \beta)$.

Both of these expressions are satisfiable, so we fork the execution. This is the case for any $k > 0$, which means that the number of possible execution

```
procedure ComputePow(a, b)
    r := 1
    i := 1
    while i ≤ b do
        r := r · a
        i := i + 1
    end while
    return r
end procedure
```

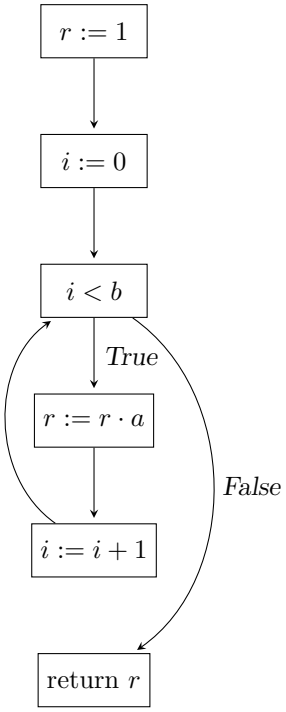

Figure 3.4: Control-flow graph for ComputePow

paths is infinite. If we insist on exploring all paths, the symbolic execution will simply continue for ever. To avoid this, we can include some other termination criteria. As an example, we could have limit on the number of times we allow the execution to fork, and as soon as this limit is reached we simply ignore any further execution paths.

3.3.2 Deciding satisfiability of path-constraints

A key component of symbolic execution, is deciding if a *path-constraint* is satisfiable, in which case the corresponding execution path is eligible for exploration. Consider the following *path-constraint* from the motivating example:

$$(\alpha \cdot 2 \geq 16) \wedge (2 \cdot \alpha - 10 \geq \beta).$$

To decide if this is satisfiable or not, we must determine if there exist an assignment of integer values to α and β such that the formula evaluates to *true*. We notice that the formula is a conjunction of linear inequalities. We can assign these to variables q_1 and q_2 and get

$$q_1 = (\alpha \cdot 2 \geq 16)$$
$$q_2 = (2 \cdot \alpha - 10 \geq \beta)$$

The formula would then be $q_1 \wedge q_2$, where q_1 and q_2 can have values *true* or *false* depending on whether or not the linear inequality holds for some integer values of α and β. The question then becomes twofold: Does there exist an assignment of *true* and *false* to q_1 and q_2 such that the formula evaluates to true? And if so, does this assignment lead to a system of linear inequalities that is satisfiable? In this example, we can assign *true* to both q_1 and q_2, which gives the following system of linear inequalities:

$$\alpha \cdot 2 \geq 16$$
$$2 \cdot \alpha - \beta \geq 10$$

where we gathered the constant terms on the right hand side, and the symbols the left hand side. We have already seen that $\alpha = 8$ and $\beta = 6$ is a satisfying assignment, so the path-constraint is indeed satisfiable. Since deciding satsifiability of a *path-constraint* is such an critical part of symbolic execution, it is important to know if we can always correctly *yes* or *no*. In the next section we will study this question more closely, and we will see that it depends on what sort of constraints we place on the input values.

The SMT problem

The example we just gave, is an instance of the *Satisfiability Modulo Theories*(SMT) problem. To understand SMT, we first consider the *Boolean Satisfiability*(SAT) problem. In this problem we are given a CNF formula , which is the logical conjunction 1 or more clauses, where a clause is the logical disjunction of one or more boolean variables or their negation. We want to decide if there exists an assignment of truth values to each variable such that the formula evaluates to true. for example, $(q_1 \vee q_2 \vee \neg q3) \wedge (q_1 \vee \neg q_2 \vee q_3)$ is a *yes*-instance of this problem, since $q_1 = true$, $q_2 = false$ and $q_3 = true$ causes the formula to evaluate to *true*. On the other hand $(q_1 \vee q_2 \vee q_3) \wedge (\neg q_1 \vee \neg q_2 \vee \neg q_3)$ is clearly a *no*-instance. This problem is *decidable*, meaning that we can always correctly

answer *yes* or *no* to whether or not a given formula is satisfiable. However it is also NP-complete, which means that we do not know any method of solving this problem with a better worst-case running time than simply trying all possible assignments which is exponential in the number of clauses (Miltersen, 2015). SMT is then an extention of this problem. In SMT, the boolean variables $q_1, q_2, \ldots, q_n$ represent expressions from some theory such as the *theory of integer linear arithmetic(LIA)* (De Moura and Bjørner, 2011). In LIA, an expression e is defined as

$$\begin{aligned}
e \in \textsc{Expression} &:= \quad p \bowtie p \\
\bowtie \in \textsc{Constraint} &:= \quad \leq \mid \geq \mid = \\
p \in \textsc{Polynomial} &:= \quad a \mid a \cdot x \mid p \circ p \\
\circ \in \textsc{Operator} &:= \quad + \mid - \\
a \in \textsc{Integer} &:= \quad 0 \mid 1 \mid -1 \mid \ldots \\
x \in \textsc{Variable} &:= \quad a \mid b \mid c \mid \ldots
\end{aligned}$$

We now want to decide if the formula is satisfiable with respect to the theory. To be satisfiable w.r.t LIA, we require that there exist an assignment of truth values to the boolean variables such that the formula evaluates to *true*, and that for such an assignment, there exists an assignment of integer values to the variables in the underlying expressions such that all the constraints are satisfied. In the previous section we saw that

$$(\alpha \cdot 2 \geq 16) \wedge (2 \cdot \alpha - 10 \geq \beta).$$

is a *yes*-instance of the SMT problem, since we could let q_1 represent $(\alpha \cdot 2 \geq 16)$ and q_2 represent $(2 \cdot \alpha - 10 \geq \beta)$. We could then assign $q_1 = true$ and $q_2 = true$. For this assignment of truth variables, we could choose $\alpha = 8$ and $\beta = 6$ in which case all the constraints hold. On the other hand

$$(2 \cdot \alpha < 4) \wedge (\alpha > 4)$$

is clearly a *no*-instance. If we let q_1 represent $(2 \cdot \alpha < 4)$ and q_2 represent $(\alpha > 4)$, the formula becomes $q_1 \wedge q_2$. The only assignment of truth variables that satisfy this is $q_1 = true$ and $q_2 = true$. But it is clear that there does not exist an integer value for α such that both the constraints hold. LIA is a *decidable* theory, meaning that we can always correctly answer *yes* or *no* to an instance of the SMT problem, when it is with respect to LIA. Given an SMT formula, we can construct an Integer Linear Program with the expressions from the formula as constraints. This program will we feasible if and only if the SMT formula is satsifiable w.r.t LIA. We can check the feasibility of the Integer linear Program by using the *branch-and-bound* algorithm. If the program is feasible, this will also give us a satisfying assignment (Vanderbei, 2001). This means that as long as we only consider programs with linear expressions as conditions, we can always decide the satisfiability of a path-constraint.

Undecidable theories

Let us consider the following extension of the definition of expressions in LIA:

$$
\begin{aligned}
e \in \textsc{Expression} &:= \quad p \bowtie p \\
\bowtie \in \textsc{Constraint} &:= \quad \leq \mid \geq \mid = \\
p \in \textsc{Polynomial} &:= \quad a \mid a \cdot x \mid p \circ p \\
\circ \in \textsc{Operator} &:= \quad + \mid - \mid \cdot \\
a \in \textsc{Integer} &:= \quad 0 \mid 1 \mid -1 \mid \ldots \\
x \in \textsc{Variable} &:= \quad a \mid b \mid c \mid \ldots
\end{aligned}
$$

This definition allows for expressions with nonlinear terms such as $5 \cdot x - 10 \leq 3 \cdot y^3$. Such expressions belong to the *Theory of Nonlinear Integer Arithmetic*(NLIA). This theory is *undecidable*, meaning that there does not exist an algorithm that always halt with a correct answer to the question of whether or not there exists an assignment of integer values to the variables such that all constraints are satisfied.

Consequences of undecidable theories for symbolic execution.

If we do symbolic execution on a program that has conditional statements with conditions that belong to an undecidable theory, we might get stuck trying to decide the satisfiability of a path constraint forever. To avoid this, we can include some other termination criteria, such as a time limit, at which point we output *unknown* instead of *yes* or *no*. This leaves us with two options. We can continue the symbolic execution and assume that the *path-constraint* is in fact satisfiable. In this case, we might end up exploring execution paths that are infeasible, meaning that no concrete input values will cause the program to execute along this path. The other option is to assume that the *path-constraint* is unsatisfiable. In this case we do not explore infeasible execution paths, but we can no longer be sure that we explore all feasible paths.

Chapter 4

Principles of concolic execution

In this chapter we will introduce *concolic execution*, which is a technique that combines concrete and symbolic execution to explore possible execution paths. We start by describing how the technique works, and then look at the advantages that it offers compared to only using symbolic execution. As in the previous chapter, we restrict our focus to programs that takes integer values as input, and performs arithmetic operations and comparisons on these.

4.1 Concolic execution of a program

During a symbolic execution of a program, we replace the inputs of the program with symbols that acts as placeholders for concrete integer values. The program environment maps variables to *symbolic values*, which can be integers, symbols or arithmetic expressions over these two. During a concolic execution of a program, we maintain two environments. One is a concrete environment M_c, which maps variables to concrete integer values. The other is a symbolic environment M_s which maps variables to symbolic values. At the beginning of the execution, the concrete environment is initialized with a random integer value for each input. The symbolic environment is initialized with symbols for each input. Finally we initialize an empty path-constraint. The program is then executed both concretely and symbolically, and at the end of this execution we determine a new set of concrete input values that will cause the concrete execution to follow a different path. In the next iteration we initialize the concrete environment with these values and the program is executed concretely and symbolically again. We continue doing this until all execution paths have been explored, or some other predefined termination criteria is met (Godefroid et al., 2005).

We will now describe what we mean by executing the program both concretely and symbolically. Specifically, we will explain how we maintain our environments, how we handle conditional statements and how we generate the next set of concrete input values.

4.1.1 Handling the environments

If we reach an assignment statement $v := e$, for some variable v and expression e, we evaluate e concretely and update our concrete environment. We also evaluate e symbolically and update our symbolic environment.

4.1.2 Handling conditional statements

Whenever we reach a conditional statement with condition q, we evaluate q concretely and chose a path accordingly. At the same time we evaluate q symbolically and get some constraint c. If the concrete value of q is true, we add c to a path-constraint. If the concrete value of q is false, we add $\neg c$ to the path constraint. This way we track which choices of paths we have made during an iteration.

4.1.3 Generating input values for the next iteration

At the end of an iteration we will have a path-constraint that, for each encountered conditional statement, describes what choice of path we made. To generate the input for the next iteration, we construct a new path-constraint by negating the condition of the last conditional statement where we have not explored the other path. We then solve the constraints from this new path-constraint to obtain the next set of input values. If some input values are not constrained by this path-constraint, we keep the current concrete value for the next iteration.

To illustrate concolic execution, we consider the program from the motivating example:
First we initialize the two environments. We let

$$M_c = \{units = 27,\ minimum = 34\}$$

where 27 and 34 is chosen randomly. We let

$$M_s = \{units = \beta,\ minimum = \alpha\}$$

where α and β are symbols. The first statement is an assignment statement, so we get two new environments:

$$M_c = \{units = 27,\ minimum = 34,\ total = 54\}$$
$$M_s = \{units = \alpha,\ minimum = \beta,\ total = 2 \cdot \alpha\}$$

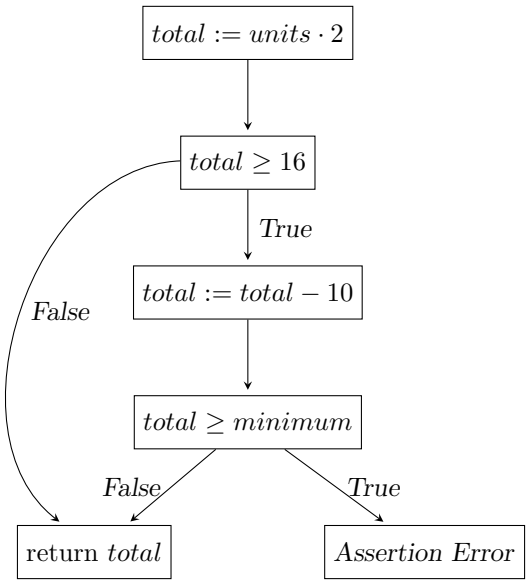

Figure 4.1: Control-flow graph for COMPUTETOTAL

Next, we reach an *if* statement with condition $total \geq 16$. Since $M_c(total) = 54$ we follow the *then* branch. We update the *path-constraint* to $[(2 \cdot \alpha \geq 16)]$. Next, we reach another *assign* statement, so we get the following environments:

$$M_c = \{units = 27,\ minimum = 34,\ total = 44\}$$
$$M_s = \{units = \alpha,\ minimum = \beta,\ total = 2 \cdot \alpha - 10\}$$

Next, we reach an *assert* statement which condition $total \geq minimum$. Since $M_c(total) = 44$ the assertion succeeds. We update the *path-constraint to* $[(2 \cdot \alpha \geq 16), ((2 \cdot \alpha - 10) \geq \beta)]$. Finally we return 44, which finishes one execution path.
To discover a new path-constraint, we make a new path-constraint by negating the final condition of the current path-constraint, so we get $[(2 \cdot \alpha \geq 16), \neg((2 \cdot \alpha - 10) \geq \beta)]$. To get the next set of concrete input values, we solve the following system of constraints

$$2 \cdot \alpha \geq 16$$
$$(2 \cdot \alpha - 10) < \beta.$$

This gives us e.g $\alpha = 8$ and $\beta = 7$. We then execute the program with $units = 8$ and $minimum = 7$. This execution will follow the same path until we reach the *assert*-statement. This time the execution results in an error due to the assertion being violated, and we have now explored all possible paths from the

assert-statement. To generate the next input values, we negate the condition from the first *if*-statement and get $[\neg(2 \cdot \alpha \geq 16)]$. From this we get e.g $\alpha = 5$. Since there are no constraints on the value of β, we keep the previous value. We now execute the program with $units = 5$ and $minimum = 7$. This execution will not follow the *then* branch in the *if*-statement, so we immediately return *total* which is 10. At this point we have explored all possible branches from each conditional statement, so we have explored all execution paths.

4.2 Handling undecidable *path-constraints*

In the previous chapter we described how symbolic execution was limited by the ability to decide satisfiability of a path-constraint. For example, if we are executing a program with inputs x and y and encounter a non linear condition $5 \cdot x - 10 \leq 3 \cdot y^3$, we might fail to decide the satisfiability of the two branches. In this case we can assume that the paths are not satisfiable and ignore them, or assume that the paths are satisfiable and continue. In the first case, we potentially miss a large number of interesting paths, and in the second case we can no longer guarantee that we only explore feasible paths.
The same issue may arise in concolic execution when generating the input values for the next iteration, but we are not left with same options as in symbolic execution. Since we always have access to both a concrete and a symbolic environment, we can avoid having non linear conditions in the path-constraint. Non linear conditions comes from arithmetic operations involving multiplication or division, where both operands contain symbols. In this case we can instead choose to evaluate the expression with the concrete environment (Godefroid et al., 2005). Consider the condition $5 \cdot x - 10 \leq 3 \cdot y^3$ again. Assume that y have concrete value 2, we then evaluate the right-hand side and get 24. The condition then becomes $5 \cdot x - 10 \leq 24$, which is within the theory of integer linear arithmetic, which we know we can solve. We can then decide satisfiability of the path with this more restrictive constraint. The cost of this is then the loss of completeness. If we decide to negate $5 \cdot x - 10 \leq 24$ and get $5 \cdot x - 10 > 24$, it is clear that solving both these constraints is not sufficient to say that all paths from the statement with condition $5 \cdot x - 10 \leq 3 \cdot y^3$ is explored. To guarantee this we would have to check for an infinite number of possible values for y. So concolic execution allows us not miss as many feasible execution paths as in symbolic execution, but we are still forced forced to give up completeness if we wish to avoid exploring infeasible paths.

Chapter 5

Introducing the *SIMPL* language

In chapter 5 and 6, we describe an implementation of a concrete and symbolic interpreter for a small toy language, written in the programming language *Scala* [1]. We do this to:

- Demonstrate the principles that we discussed in chapter 3, by describing an actual implementation of symbolic execution for a small toy language.
- Be able to clearly see the differences between a symbolic and a concrete execution by comparing the two interpreters, and looking at the necessary changes to the source language.

In this chapter we introduce the toy language, called *SIMPL* . We start by giving a formal definition through a grammar, and then we describe a purely functional implementation of a concrete interpreter for the language. We have chosen to implement both interpreters in a purely functional way to avoid dealing with states and side effects.This will allow us to easily see the differences by simply comparing relevant function signatures. In order to do this, we also describe a technique to avoid throwing exception whenever we encounter errors, such as type errors, assertion errors or referencing undefined variables or functions. Doing this gives a much more elegant implementation, and it greatly simplifies the implementation of the symbolic interpreter as we do not wish to interrupt the execution whenever we encounter such an error.

5.1 Syntax of *SIMPL*

SIMPL consists of expressions and statements. Expressions evaluates to values and do not change the control flow of the program. A statement evaluates to a

[1] Full implementation can be found on GitHub at `https://git.io/fj2Np`.

value and a possibly updated variable environment. They may also change the control flow of the program through conditional statements.

5.1.1 Expressions

Expressions consists of concrete values which can be integers, booleans, and a special *unit* value. Furthermore they consist of variables referenced by identifiers. Finally they consist of arithmetic operations on integers and comparison of integers

$$
\begin{array}{rll}
int \in \textsc{Integer} & := & 0 \mid 1 \mid -1 \mid \ldots \\
bool \in \textsc{Boolean} & := & True \mid False \\
val \in \textsc{ConcreteValue} & := & int \mid bool \mid unit \\
id \in \textsc{Identifier} & := & a \mid b \mid c \mid \ldots \\
exp \in \textsc{Expression} & := & val \mid id \\
& \mid & exp + exp \mid exp - exp \mid exp * exp \mid exp/exp \\
& \mid & exp < exp \mid exp > exp \mid \quad exp \leq exp \\
& \mid & exp \geq exp \mid exp == exp
\end{array}
$$

5.1.2 Statements

Statements in *SIMPL* consists of assignments to values. Variables are implicitly declared, so assigning to a variable that does not already exist, will create the variable.

$$
smt \in \textsc{Statement} := \quad id = exp
$$

The value of an assignment statement is the value of the expression on the right hand side. This expression must evaluate to either an integer value or a boolean value. *SIMPL* supports three different conditional statements; *if*-statements, *while*-statements and *while*-statements.

$$
\begin{array}{rll}
stm \in \textsc{Statement} & := & \textbf{if } exp \textbf{ then } stm \textbf{ else } stm \\
& \mid & \textbf{while } exp \textbf{ do } stm \\
& \mid & \textbf{assert } exp
\end{array}
$$

The condition expression must evaluate to a boolean value. The value of an *if*-statement is the value of the statement that gets evaluated depending on the value of the condition. The value of a *while*-statement is *unit*. The value of an *assert*-statement is *unit* if the condition evaluates to $True$, otherwise it is an assertion error. Finally a statement can simply be an expression or a sequence of statements.

$$stm \in \textsc{Statement} := \quad exp \mid stm\ stm$$

5.1.3 Functions and programs

SIMPL supports top-level functions that must be defined at the beginning of the program. A function definition consists of an identifier, followed by a parameter list with zero or more identifiers. The function body consists of one or more statements. A function call is an expression consisting of an identifier matching one of the defined functions followed by a argument list with zero or more expresssions. The value of a function call is the value of the final statement in the function body.

$$\begin{aligned} f \in \textsc{Function} &:= \quad id\ (id^*)\ \{stm\} \\ call \in \textsc{Expression} &:= \quad id\ (exp^*) \end{aligned}$$

Since expressions only evaluates to a value, it follows that function calls does not have any side effects.

Finally we define the syntax of a *SIMPL* program to be one or more function declarations, followed by a call to one of these functions.

$$prog \in \textsc{Program} := \quad f\ f^*\ call$$

5.2 Concrete interpreter for *SIMPL*

To interpret *SIMPL* we define the three following functions

```
def interpProg(p: Prog): Result[ConcreteValue, String]

def interpExp(p: Prog, e: Exp, env: Map[Id, ConcreteValue]):
    Result[ConcreteValue, String]

def interpStm(p: Prog, s: Stm, env: Map[Id, ConcreteValue]):
    Result[(ConcreteValue, Map[Id, ConcreteValue]), String]
```

Figure 5.1: High level overview of the interpreter implementation in *Scala*.

where *interpProg* acts as the entry function. It takes a program p of type *Prog(funcs: Map[Id, FDecl], fCall: CallExp)* and calls *interpExp* with p, *fCall* and a fresh environment. We use an immutable map of type

Map[Id, ConcreteValue]

to represent our environment. From the signature of *interpExp* we see that the interpretation of an expression depends on the current environment and the program which holds the top level functions, and that it results in a concrete value. The final function is *interpStm* whose signature tells us that the interpretation of a statement also depends on the current environment and the program, and that results in both a concrete value and a potentially updated environment.

5.2.1 Error handling without side effects

As previously mentioned we wish to avoid side effects in our implementation. We must therefore avoid throwing exceptions whenever we encounter errors. Instead we define a trate of type

$$Result[+V, +E]$$

that can either be *Ok(v: V)*, or *Error(e: E)*, for some types V and E (Chiusano and Bjarnason, 2014).

```
trait Result[+V, +E]
case class Ok[V](v: V) extends Result[V, Nothing]
case class Error[E](e: E) extends Result[Nothing, E]
```

We let *InterpExp* have return value *Result[ConcreteValue, String]*, meaning we return *Ok(v: ConcreteValue)* if we do not encounter any errors, and *Error(e: String)* if we do. In this case e will be a message that describes what sort of error we encountered. *InterpStm* has return value *Result[(ConcreteValue, HashMap[Id, String]), String]* since we also return a potentially updated environment.

Map, flatMap and traverse

We define three functions *map*, *flatMap* and *traverse*, to be able to apply functions to the results:

```
def map[T](f: V => T): Result[T, E] = this match {
case Ok(v) => Ok(f(v))
case Error(e) => Error(e)
}

def flatMap[EE >: E, T](f: V => Result[T, EE]): Result[T, EE]
= this match {
   case Ok(v) => f(v)
   case Error(e) => Error(e)
}

def traverse[V, W, E](vs: List[V])(f: V => Result[W, E]):
 Result[List[W], E] = vs match {
```

```
    case Nil => Ok(Nil)
    case hd::tl => f(hd).flatMap( w => traverse(tl)(f)
.map(w :: _))
}
```

For some result r, *map* allows us to apply a function $f : V \to T$ to v if $r = Ok(v)$. Otherwise we simply return r. *flatMap* has the same functionality except that we apply a function that also returns a *Result*. This way we avoid nesting like *Ok(Ok(v))*. These two functions allows us to handle errors seamlessly. If, for example we wish to interpret an arithmetic expression *AExp(e1, e2, Add())*, we simply do

```
interpExp(p, e1, env).flatMap(
   v1 => interpExp(p, e2, env).map(v2 => v1.v + v2.v)
)
```

If we encounter an error during the interpretation of *e1* this error is returned immediately. If not we continue by interpreting *e2*. If we encounter an error here, we return this error, otherwise we continue and return the sum of the two expressions. Here we assume that *v1* and *v2* are of type *IntValue(v: Int)*. The full implementation also includes checking that both expressions evaluate to the proper types.
Finally we define a function *traverse*, that takes a list of type V, a function $f : V \to Result[W, E]$ and returns a *Result[List[W], E]*. We use this function to interpret functions calls *CallExp(Id, List[Exp])*. To do this, we need to evaluate the argument list, and if we do not encounter any errors, construct a local environment and evaluate the function body. We could simply map *interpExp* on to the argument list, and check for each element in the resulting list if it is an error. This would require two passes over the list. Instead, *traverse* applies f to each element in the list, and if it results in an error, we immediately return this error. Otherwise we prepend the resulting value onto the result of traversing the remaining list. We only pass over the list once, and we immediately return the first error encountered.

Chapter 6

Symbolic execution of *SIMPL*

In this chapter we will describe a symbolic interpreter for *SIMPL* . First, we describe how we must extend the grammar, to be able to do symbolic execution. Next we describe the implementation of *interpExp* and *interpStm*, and how these relate to their concrete counterparts.

6.1 Extension of grammar

The original language have three types of values, namely integers, booleans and the unit value. Symbolic execution requires that we extend the language to include symbolic counterparts of these. We therefore introduce a symbolic value which can either be a symbolic integer($\widehat{int}$), a symbolic boolean($\widehat{bool}$) or the unit value.

$$val \in \textsc{SymbolicValue} := \quad \widehat{int} \mid \widehat{bool} \mid unit$$

A symbolic integer can either be a concrete integer, a symbol or an arithmetic expression over these two:

$$\begin{aligned}
int \in \textsc{Integer} &:= \quad 0 \mid 1 \mid -1 \mid \ldots \\
sym \in \textsc{Symbol} &:= \quad a \mid b \mid c \mid \ldots \\
\widehat{int} \in \textsc{SymbolicInteger} &:= \quad int \mid sym \mid \widehat{int} + \widehat{int} \\
&\mid \quad \widehat{int} - \widehat{int} \mid \widehat{int} * \widehat{int} \\
&\mid \quad \widehat{int} / \widehat{int}
\end{aligned}$$

A symbolic boolean can either be $True$, $False$ or an symbolic boolean expression, which is a comparison of two symbolic integers. Finally a symbolic boolean can be the negation of a symbolic boolean. This extension is needed to be able to represent the two different *path-constraints* that might arise from a conditional statement.

$$
\begin{aligned}
bool \in \textsc{Boolean} &:= True \mid False \\
\widehat{bool} \in \textsc{SymbolicBoolean} &:= \widehat{int} < \widehat{int} \mid \widehat{int} > \widehat{int} \\
&\mid \widehat{int} \leq \widehat{int} \mid \widehat{int} \geq \widehat{int} \\
&\mid \widehat{int} == \widehat{int} \mid ! \ \widehat{bool}
\end{aligned}
$$

6.2 Path-constraints

To represent a path-constraint, we implement the following classes:

```
case class PathConstraint(conds: List[SymbolicBool], ps: PathStatus) {
     def :+(b: SymbolicBool): PathConstraint =
         PathConstraint(this.conds :+ b, this.ps)
}
sealed trait PathResult
case class Certain() extends PathResult
case class Unknown() extends PathResult
```

The definition of *PathConstraint* consists of two elements. *conds* is a list of symbolic booleans from the conditional statements met so far. The *PathStatus* tells us whether or not we can guarantee that the path constraint is in fact satisfiable. This is necessary because we allow for nonlinear constraints, which our SMT solver may fail to solve. In the case of a failure, we still explore the path but the status of the path-constraint will be *Unknown*, meaning that we cannot guarantee that there exists concrete input values that follow this path.

6.3 Symbolic interpretation of expressions

To symbolically interpret an expression, we define the following function illustrated in figure 6.1:

```
     def interpExp(p: Prog, e: Exp, env: Map[Id, SymbolicValue],
          pc: PathConstraint, forks: Int):
             List[(Result[SymbolicValue, String], PathConstraint)]
```

Figure 6.1: Symbolic version of *InterpExp*

```
def interpExp(p: Prog, e: Exp, env: Map[Id, ConcreteValue]):
    Result[ConcreteValue, String]
```

Figure 6.2: Concrete version of *InterpExp*

By comparing figure 6.1 with figure 6.2, we see that the symbolic version of *interpExp* is an extension of the version from the concrete interpreter. It still depends current environment, which is now of type *Map[Id, SymbolicValue]*, and the program itself. However, it now also depends on a path-constraint to keep track on any constraints placed on the symbolic values, and *forks* which is the current number of times the execution has been forked. This value is used to enforce an upper bound on the number of execution paths that we explore to avoid running forever.

The return type has also changed. Instead of returning a single result, it returns a list of pairs consisting of a result and a path-constraint, one for each possible execution path. This also effects how we interpret the different types of expressions. Most of them consists of one or more sub-expressions, each of which may result in several different results, so we have to consider all possible combinations of results. We will demonstrate this by looking at the interpretation of arithmetic expressions and function calls.

6.3.1 Arithmetic expressions

Consider an arithmetic expression *AExp(e1: Exp, e2: Exp, op: AOp)*. When we recursively interpret e_1 and e_2, we get two lists L_{e_1}, L_{e_2} of possible results. We need to take the cartesian $L_{e_1} \times L_{e_2}$ of the lists, and for each pair of results, we have to evaluate the arithmetic expression.

To do this we use a *for-comprehension* which given two lists, iterate over each ordered pair of elements from the two lists. For each pair, we *flatMap* over the two results, and if no errors are encountered, we check that both expressions evaluates to symbolic integers and compute the result. If both sub expression evaluate to concrete integers, we compute the result as we would do in a concrete interpretation.

6.3.2 Function calls

Consider a *Call*-expression *CallExp(id: Id, args: List[Exp])*. First, we must check that the function is defined and that the formal and actual argument list does not differ in length. If both of these checks out, we map *interpExp* on to *args*. This gives a list of lists $[L_{e_1}, L_{e_2}, \ldots, L_{e_n}]$, where the $i'th$ list contains the possible results of interpreting the $i'th$ expression in *args*. We take the cartesian product $L_{e_1} \times L_{e_2} \times \ldots \times L_{e_n}$, which gives us all possible argument lists. For each of these lists, we zip it with formal argument list. This gives a list of the following type *List[(ExpRes, Id)]*. We attempt to build a local environment by

folding over this list, starting with the original environment that was passed with the call to *interpExp*. If we encounter an error, either from the interpretation of the expressions in *args* or from an expression evaluating to the unit value, the result of the function call with this argument list will be that error. Otherwise, for each pair of result and id, we make a new environment with the appropriate binding added. We then interpret the statement in the function body with the local environment, and the result of the function call with this argument list, will be the value of the statement.

6.4 Symbolic interpretation of statements

To symbolically interpret statements, we define the following function shown in figure 6.3:

```
def interpStm(p: Prog, stm: Stm, env: Map[Id,SymbolicValue],
  pc: PathConstraint, forks: Int):
    List[(Result[SymbolicValue, String], Map[Id, SymbolicValue],
        PathConstraint)]
```

Figure 6.3: Symbolic version of *interpStm*

```
def interpStm(p: Prog, s: Stm, env: Map[Id, ConcreteValue]):
    Result[(ConcreteValue, Map[Id, ConcreteValue]), String]
```

Figure 6.4: Concrete version of *interpStm*

By comparing figure 6.3 and figure 6.4 we see that this version of *interpStm* is also an extension of its concrete counterpart in the same way that the symbolic version of *interpExp* is. It now also depend on a path-constraint and the number of forked executions. Similarly the return type has changed so that it now returns a list of triples consisting of a result, a path-constraint and a potentially updated environment. We will now demonstrate how we interpret conditional statements. We will also see how we deal with undecidable *path-constraints* and how we limit the number of paths that we explore.

6.4.1 Conditional statements

As mentioned earlier, *SIMPL* supports three different types of conditional statements, which are represented by the following classes:

```
case class IfStm(cond: Exp, thenStm: Stm, elseStm: Stm)
case class WhileStm(cond: Exp, doStm: Stm)
case class AssertStm(cond: Exp)
```

For each possible result of interpreting the condition expression we do the following: If it is either *True()* or *False()*, the expression must have been free from symbols, so we can continue as we would have done during a concrete interpretation, with the same path-constraint and value of *forks*.
If the result is a symbolic boolean expression b, we have two potential paths to follow. We must check the satisfiability of *pc :+ b* and *pc :+ Not(b)*. We do this by calling *(checkSat(pc, b)* and *checkSat(pc, Not(b))*. If *checkSat* reports *SATISFIABLE*, the given path is eligible for exploration. If it reports *UNSATISFIABLE*, we can safely ignore the path. If it reports *UNKNOWN*, we still explore the path, but the new path-constraint will have its status set to *Unknown*. If both calls to *checksat* return either *SATISFIABLE* or *UNKNOWN*, we explore both paths, and return a list containing the results of the first path, followed by the results of the second path.

Checking satisfiability

To check the satisfiability of a path, we define the following function

```
def checkSat(pc: PathConstraint, b: SymbolicBool): z3.Status
```

which takes a path-constraint and a symbolic boolean b, and checks the satisfiability of $b_1 \wedge b_2 \wedge \ldots \wedge b_k \wedge b$, where $b_1, b_2, \ldots, b_k$ are the symbolic booleans in *pc.conds*. To do this, we use the *Java* implementation of the *Z3* SMT solver.

Uppper bound on number of forks

The interpretation of an *If*-statement and a *While*-statement starts with checking whether $forks > maxForks$, in which case we immediately return an Error. We include this check to prevent the interpreter to run forever on programs with an infinite number of execution paths.

Chapter 7

Related work

Classical symbolic execution is introduced in (King, 1976). The paper establishes the concept of symbolically executing a program by replacing concrete input values by symbols, and letting variables reference integer polynomials over these symbols. The concept of a *path-constraint* is also introduced as a conjunction of boolean expressions over the input symbols. These expressions are constraints placed on the input symbols that must be satisfied to execute a long the given path. Whenever the execution reaches a conditional statement, e.g an *if*-statement, the satisfiability of both execution path is checked. If both paths are satisfiable, the execution is forked into two independent executions. One execution follows the path corresponding to the condition evaluating to true, while the other follows the path corresponding to the condition evaluating to false. The paper also touches on the difficulty of deciding the satisfiability of paths, and that for most practical programs the total number of execution paths are too large to systematically explore. (Godefroid et al., 2005) presents a tool called DART(Directed Automated Random Testing), which is an instance of concolic execution. The program is executed with initially random input values. During the execution, both a concrete and a symbolic environment is maintained, and whenever the execution branches, the choice of branch is registered in a path-constraint , using the symbolic environment. At the end of an execution, the path-constraint is used to generate a new set of concrete that follow a different execution path. One of the main advantages of the DART approach is the ability to avoid getting stuck trying to decide the satisfiability of a *path-constraint* . This is achieved by substituting in concrete values whenever such undecidable contraints arises.

Chapter 8

Conclusion

We have demonstrated how symbolic execution, in principle, allows us to explore all possible execution paths for a program by executing the program using *symbolic* values instead of concrete concrete input values. By keeping track of any constraints placed on these values by following a given execution path, we can solve these constraints to obtain concrete input values that follows the same path. This technique does have some limitations. One such limitation is the path explosion problem. The number of possible paths is potentially exponential in the number of conditional statements, and there might even be an infinite number of paths. Another limitation is the ability to decide whether a set of constraints is satisfiable or not. For some types of constraints this question is undecidable, and in this case we either have to give up exploring a large number of execution paths, or risk exploring paths that will never be executed by any concrete input values. We have also seen how this last limitation is partially mitigated by *concolic execution*, where we combine concrete and symbolic execution. Here the program is executed with initially random concrete input values. During execution we maintain both symbolic and concrete values for the variables, and whenever the program branches, we keep track of which branch was chosen in a *path-constraint*. This path-constraint is then used to generate the input for the next execution, which will follow a different path. By having access to both concrete and symbolic values, we can substitute symbolic values with concrete values if we ever encounter a path-constraint that we cannot solve. While this allows us to explore more paths without exploring infeasible paths, we still have to give up completeness, so the problem is not fully mitigated.
Finally we demonstrated symbolic execution by implementing both a concrete and a symbolic interpreter for a small toy language, which we compared to clearly see the differences between a concrete and symbolic execution and how the source language was extended to allow for symbolic execution.

Bibliography

Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56:82–90, 02 2013. doi: 10.1145/2408776.2408795.

Paul Chiusano and Rnar Bjarnason. *Functional Programming in Scala*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2014. ISBN 1617290653, 9781617290657.

Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011. ISSN 0001-0782. doi: 10.1145/1995376.1995394. URL `http://doi.acm.org/10.1145/1995376.1995394`.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005. ISSN 0362-1340. doi: 10.1145/1064978.1065036. URL `http://doi.acm.org/10.1145/1064978.1065036`.

James C. King. Symbolic execution and program testing. *Commun. ACM*, 19 (7):385–394, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL `http://doi.acm.org/10.1145/360248.360252`.

Peter Bro Miltersen. *P, NP and NPC*. 3rd edition, 2015.

Robert J. Vanderbei. Linear programming: Foundations and extensions, 2001.