Symbolic Execution(Working title)

Aarhus Universitet

Søren Baadsgaard

May 9, 2019

Abstract

Contents

1 Introduction **3**

2 Motivation **4**
2.1 A motivating example 4

3 Principles of symbolic execution **6**
3.1 Symbolic executing of a program 6
3.2 Execution paths and path constraints 7
3.3 Constraint solving 10
3.4 Limitations and challenges of symbolic execution 10
3.4.1 The number of possible execution paths 10
3.4.2 Deciding satisfiability of *path-constraints* 12
3.4.3 Undecidable theories 13

4 Principles of Concolic execution **15**
4.1 Concolic execution of a program 15
4.1.1 Assignment statements 16
4.1.2 Conditional statements 16
4.1.3 Generation of concrete input values 16
4.2 Handling undecidable *path-constraints* 18

5 Introducing the language *SIMPL* **19**
5.1 Syntax of *SIMPL* 19
5.1.1 Expressions 19
5.1.2 Statements 20
5.2 Concrete interpreter for *SIMPL* 21
5.2.1 Error handling 22

6 Symbolic execution of *SIMPL* **24**
6.1 Extension of grammar 24
6.2 Path-constraints 25
6.3 Interpretation of expressions 25
6.3.1 Arithmetic and boolean expressions 25
6.3.2 Function calls 27

7 Concolic execution of *SIMPL* **29**

8 Conclusion **30**

A Source code **31**

B Figures **32**

Chapter 1

Introduction

Chapter 2

Motivation

In this chapter we will present the motivation for this project, by considering 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 with a unit price 2. If the revenue of an order is greater than or equal 16, a discount of 10 is applied. The following program that takes integer inputs *units* and *cost* computes the total revenue based on this pricing scheme.

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

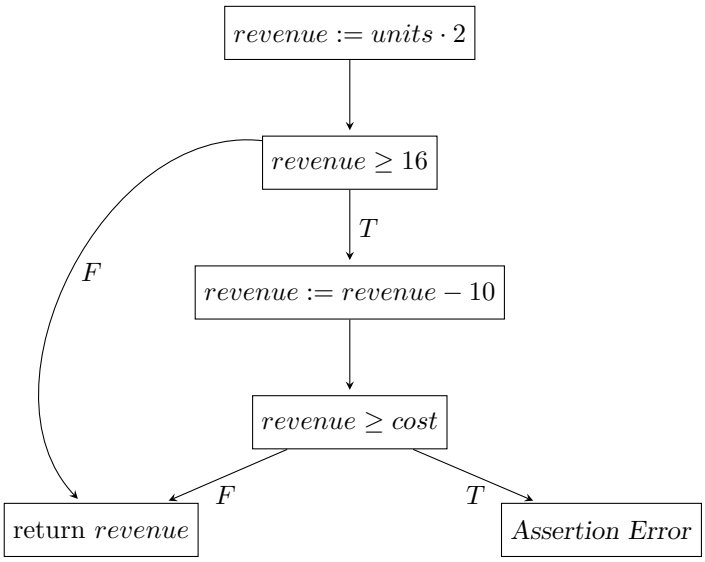

Figure 2.1: Control-flow graph for ComputeRevenue

We assert that $revenue \geq cost$ if we apply the discount, since we do not wish to sell at a loss. 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. We might try to run the program on different input values, e.g $(units = 8, cost = 5)$, $(units = 7, cost = 10)$. 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 observing the program for some time, we realize that the input must satisfy the following two constraints to fail:

$$units \cdot 2 \geq 16$$
$$units \cdot 2 < cost$$

which is the case e.g for $(units = 8, cost = 7)$. 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*. The program environment consists of variables that can reference both concrete and symbolic values. [1].
To illustrate this, we consider the following program that takes parameters a, b, c and computes the sum:

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

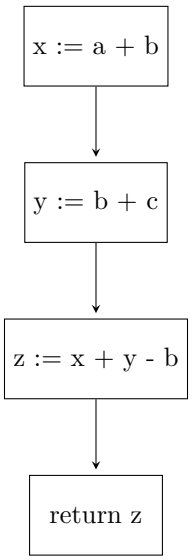

Figure 3.1: Control-flow graph for ComputeSum

Lets consider running the program with concrete values $a = 2, b = 3, c = 4$. 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 would 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* only contains the expression $true$, since we have not encountered any conditional statements. to continue execution along a path,

$q_1 \wedge \ldots \wedge q_k$ must *satisfiable*. To be *satisfiable*, there must exist an assignment of integers to the symbols, such that the conjunction of the expressions 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*.
Whenever we reach a conditional statement with condition q_k, we consider the two following expressions:

1. $pc \wedge q_k$
2. $pc \wedge \neg q_k$

where pc is the conjunction of all the expressions currently contained in the *path-constraint*.
This gives a number of possible scenarios:

- **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 variable state and *path-constraints* that are the same up to the final element. One will have q_k as the final element 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.

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

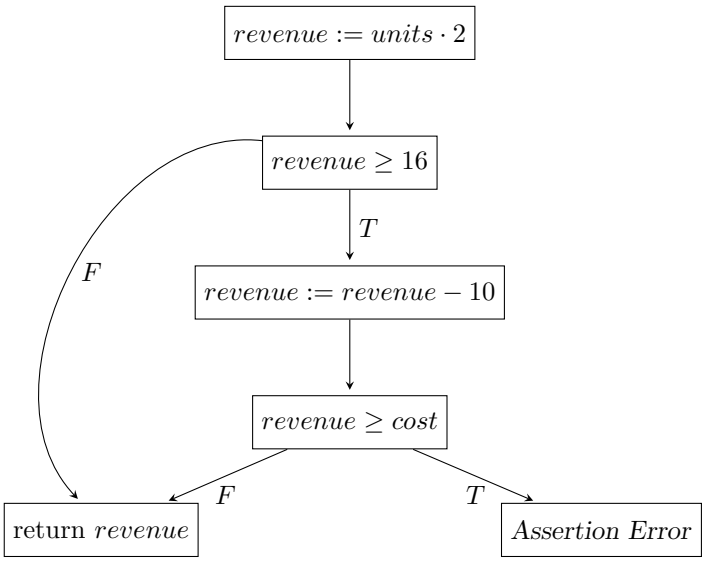

Figure 3.2: Control-flow graph for ComputeRevenue

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

First we assign $2 \cdot \alpha$ to *revenue*. We then reach a conditional statement with condition $q_1 = \alpha \cdot 2 \geq 16$. To proceed, we need to check the satisfiability of the following two expressions:

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

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

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

Both of these expressions are satisfiable, so we fork again. In the end we have discovered all three possible execution paths:

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

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

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

The first two *path-constraints* represents 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, does not result in a crash. The final *path-constraint* represents 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.

3.3 Constraint solving

In the previous section we described how to handle programs with multiple execution paths by introducing a *path-constraint* for each path, which a list of constraints on the input symbols. This system of constraints defines a class of integers that will cause the program to execute along this path. By solving the system from each *path-constraint*, we obtain a member from each of class which forms a set of concrete inputs that cover all possible paths.

If we consider the motivating example again, we found three different paths, represented by the following *path-constraints*:

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

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

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

By solving for α and β, we obtain the set of inputs $\{(7, \beta), (8, 6), (8, 7)\}$, that covers all possible execution paths. Note that we have excluded a concrete value for β in the first test case, because the *path-constraint* does not depend on the value of β.

3.4 Limitations and challenges 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. In this section we will cover the challenges that arise when we consider more general programs.

3.4.1 The number of possible execution paths

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. 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:

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

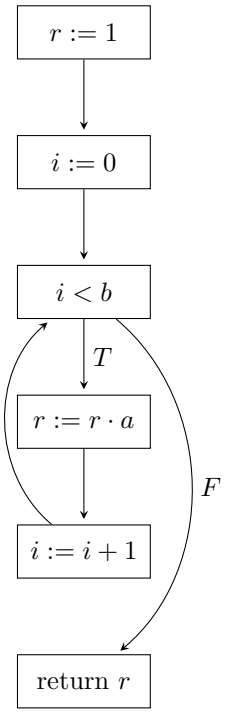

Figure 3.3: Control-flow graph for ComputePow

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

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

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 paths is infinite. If we insist on exploring all paths, the symbolic execution will simply continue for ever.

3.4.2 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:

$$true \wedge (\alpha \cdot 2 \geq 16) \wedge \neg(2 \cdot \alpha - 10 < \beta). \tag{3.1}$$

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

$$q_1 = (\alpha \cdot 2 \geq 16) \tag{3.2}$$
$$q_2 = (2 \cdot \alpha - 10 < \beta) \tag{3.3}$$

The formula would then be $true \wedge q_1 \wedge \neg q_2$, 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 q_1 and *false* q_2, which gives the following system of linear inequalities:

$$\alpha \cdot 2 \geq 16 \tag{3.4}$$
$$2 \cdot \alpha - \beta \geq 10 \tag{3.5}$$

where we gathered the constant terms on the left hand side, and the symbols the right hand side. From the first equation we get that $\alpha \geq 8$so we select $\alpha = 8$. From the second equation we then get that $\beta \leq 6$, so we select $\beta = 6$ and this gives us a satisfying assignment for the path constraint.

The SMT problem

The example we just gave, is an instance of the *Satisfiability Modulo Theories(**SMT**) probem*. In this problem we are given a logical formula over boolean variables $q_1, q_2, \ldots, q_n$, or their negation. The task is then to decide if there exist an assignment of boolean values $true$ and $false$ to this variables, so that the formula evaluates to *true*. Furthermore, each of these boolean variables represent some formula belonging to a theory. Such a theory could be the *theory*

*of Linear Integer Arithmetic(**LIA**)* which we will explain shortly. If there exist an assignment that satisfies the original formula, this assignment must also be valid w.r.t the given theory. Note that the first part of this problem is simply the *boolean SAT problem*, which is known to be *NP-complete*, so solving this part alone takes worst-case exponential time.

The theory of linear integer arithmetic

The conditions that we have studied so far, have all had the following form:

$$
\begin{aligned}
&a_0 + a_1 \cdot x_1 + a_2 \cdot x_2 + \ldots + a_n \cdot x_n \bowtie b \\
&where \\
&\bowtie \in \{\leq, \geq, =\} \\
&x_1, \ldots, x_n \in \mathbb{Z}
\end{aligned}
$$

which is exactly the atomic expressions in the *theory of linear integer arithmetic* **(LIA)**.

As an example, consider the following *path-constraint* again:

$$true \wedge (\alpha \cdot 2 \geq 16) \wedge \neg(2 \cdot \alpha - 10 < \beta). \tag{3.6}$$

We can express this as the **SMT** formula $true \wedge q_1 \wedge \neg q_2$ with $q_1 = (\alpha \cdot 2 \geq 16)$ and $q_2 = (2 \cdot \alpha - \beta < 10)$, where q_1 and q_2 are atomic expressions of **LIA**.

An important property of **LIA** is the fact that it is decidable. Given a formula over a number of atomic expressions, we can construct a *Integer Linear Program* **(ILP)** with these expressions as constraints, and a constant objective function. This **ILP** is feasible if and only if the formula is satisfiable, and we can check the feasibility by using the *branch-and-bound* algorithm.

3.4.3 Undecidable theories

We just saw that the conditions we have considered so far, are atomic expressions in the *Theory of Linear Integer Arithmetic*, and that this theory is decidable. This means that we can always decide whether a given execution path is eligible for exploration.

Lets consider the following extension of the conditions that we can encounter:

$$
\begin{aligned}
&a_0 \circ a_1 \cdot x_1 \circ a_2 \cdot x_2 \circ \ldots \circ a_n \cdot x_n \bowtie b \\
&where \\
&\bowtie \in \{\leq, \geq, =\} \\
&\circ \in \{+, \cdot\} \\
&x_1, \ldots, x_n \in \mathbb{Z}
\end{aligned}
$$

This allows for non linear constraints such as $3 \cdot \alpha^3 - 7 \cdot \beta^5 \leq 11$. Such expressions does not belong to **LIA**, so we are no longer guaranteed that we can decide satisfiability of the *path-constraints*. In fact, they belong to the *Theory of Nonlinear Integer Arithmetic* which has been shown to be an undecidable theory. This presents us with a major limitation of symbolic execution, since we might get stuck trying do decide the satisfiability of a *path-constraint* that is not decidable.

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 value. The program state consists of variables that reference *symbolic values*, which can either 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. The program is then iteratively executed both concretely and symbolically. At the end of each iteration, we try do determine a new set of input values, that will cause the program to follow a different execution path in the next iteration. We do this until all execution paths have been explored, or some other predefined termination criteria is met. We will now describe what we mean by execution 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 Assignment statements

If we reach an assignment statement $x := e$, 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 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 track which choices of paths we have made during an iteration.

4.1.3 Generation of concrete input values

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 a new set of input values, we make a new *path-constraint*by negating the final condition in the original *path-constraint*. That is, if we end up with $pc = [c_1, c_2, \ldots, c_n]$, we make a new *path-constraint*$pc' = [c_1, c_2, \ldots, \neg c_n]$. We then solve the system of constraints from this *path-constraint*to get new concrete input values. If some input values were constrained by the *path-constraint*, we can simply keep these values for the next run.

To illustrate concolic execution, we examine the program from the motivating example:

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

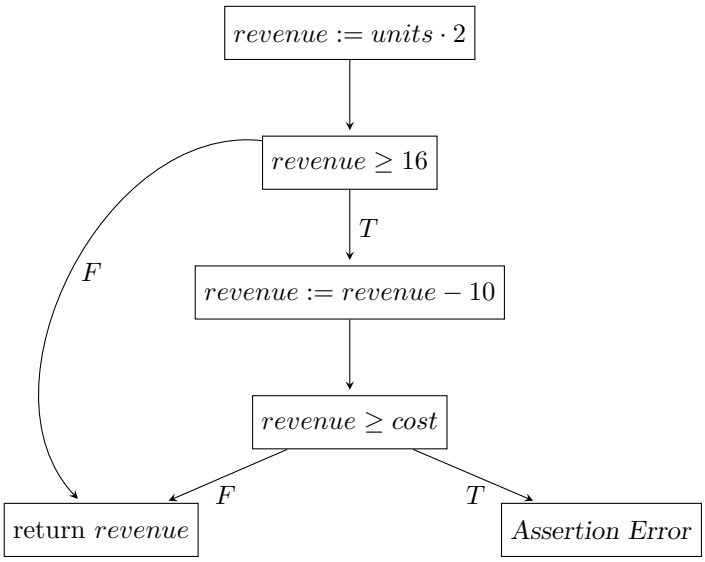

Figure 4.1: Control-flow graph for ComputeRevenue

First we initialize the two environments. We let

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

where 27 and 34 is chosen randomly. We let

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

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

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

Next, we reach an *if* statement with condition $revenue \geq 16$. Since $M_c(revenue) = 54$ we follow the *then* branch. At the same time we get a new *path-constraint* which is $[(2 \cdot \alpha \geq 16)]$, since we follow the *then* branch. Next, we reach another *assign* statement, so we get the following environments:

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

Next, we reach an *assert* statement which condition $revenue \geq cost$. Since $M_c(revenue) = 44$ the assertion succeeds. This gives us a new *path-constraint*

$[(2 \cdot \alpha \geq 16), ((2 \cdot \alpha - 10) \geq \beta)]$. 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, M_c we solve the system of constraints given by this *path-constraint*

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

This gives us e.g $\alpha = 8$ and $\beta = 7$. We then re execute the program with $units = 8$ and $cost = 7$. This execution will follow the same path until we reach the *assert*-statement, since we generated the current input by solving for the negation of the *assert* condition. This time the execution results in an error, due to the assertion being violated. Since we have now explored both possible outcomes of the *assert* statement, we make a new *path-constraint*that just contains the negation of the first condition. So we get $[\neg(2 \cdot \alpha \geq 16)]$. From this we get e.g $\alpha = 5$. We now re execute the program with $units = 5$ and $cost = 11$, where 11 was chosen randomly. This execution will not folow the *then* branch in the *if* statement, so we immediately return $revenue$ which is 10. At this point we have executed each possible outcomes of each conditional statement so have explored all possible execution paths. Finally we have generated the inputs $\{(27, 34), (8, 7), (5, 11)\}$ that cover all possible 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 execution 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 either let the execution fail, or assume that the paths are satisfiable and continue. In the first case, we potentially miss a large number of possible paths, and in the second case we can no longer guarantee that the result we return is sound.
The same issue may arise in concolic execution when trying to generate the concrete input values for the next iteration. If we consider the same program again, and we fail to solve $5 \cdot x - 10 \leq 3 \cdot y^3$, we do not have to give up. Since we have both a concrete and a symbolic environment, we can replace a symbolic value with a concrete one. If we assume that y has concrete value 2, we can insert this and get $5 \cdot \alpha - 10 \leq 24$. This is a linear constraint, which we can solve, so we can continue the execution. This of course means that we do not necessarily discover all possible execution paths, since there might exist an execution path with *path-constraint*$[(5 \cdot x - 10 \leq 3 \cdot y^3), (y > 49)]$. This is still an improvement over completely aborting the execution or giving up soundness.

Chapter 5

Introducing the language *SIMPL*

In this chapter we will introduce *SIMPL* which is a small programming language which consists of top-level functions, expressions and statements. We start by describing the syntax of the language, and then we describe a concrete interpreter for the language.

5.1 Syntax of *SIMPL*

SIMPL consists of expressions and statements. Expressions evaluates to values and does not change the control flow of the program. A statement evaluates to a 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$\langle I, \rangle$, booleans$\langle B \rangle$, and a special *unit* value. Furthermore they consist of variables that are referenced by identifiers$\langle Id \rangle$. Finally they consist of arithmetic operations and comparisons of integers.

$\langle I \rangle$ ::= 0 | 1 | -1 | 2 | -2 | ...

$\langle B \rangle$ ::= True | False

$\langle CV \rangle$ ::= $\langle I \rangle$
| $\langle B \rangle$
| unit

$\langle Id \rangle$ ::= a | b | c | ...

$\langle E \rangle$ ::= $\langle CV \rangle$
  | $\langle E \rangle$ + $\langle E \rangle$ | $\langle E \rangle$ - $\langle E \rangle$ | $\langle E \rangle$ * $\langle E \rangle$ | $\langle E \rangle$ / $\langle E \rangle$
  | $\langle E \rangle$ <$\langle E \rangle$ | $\langle E \rangle$ >$\langle E \rangle$ | $\langle E \rangle \leq \langle E \rangle$ | $\langle E \rangle \geq \langle E \rangle$ | $\langle E \rangle$ == $\langle E \rangle$

5.1.2 Statements

variable declaration and assignment

Variables implicitly declared, so variable declaration and assignment are contained in the same expression:

$\langle S \rangle$ ::= $\langle Id \rangle = \langle Exp \rangle$

The value of an *assignment*-statement is the value of the expression on the right-hand side.

Conditional statements

SIMPL supports three different conditional statements, namely *if-then-else* statements, *while* statements and *assert* statements:

$\langle S \rangle$ ::= if $\langle E \rangle$ then $\langle S \rangle$ else $\langle S \rangle$
  | while $\langle E \rangle$ do $\langle S \rangle$
  | assert $\langle E \rangle$

Where the condition must be an expression that evaluates to a boolean value. The value of an *if-then-else* statement is the value of the statement that ends up being evaluated, depending on the condition. In a *while* statement, we are not guaranteed that the second statement is evaluated, so we introduce a special *unit* value which will be the value of any *while* statement. An assert statement will have the *unit* value if the condition evaluates to *true*. If the condition evaluates to *false*, the execution ends with an error.

Finally a statement may simply be an expression, or one statement followed by another:

$\langle S \rangle$ ::= $\langle E \rangle$
  | $\langle S \rangle$ $\langle S \rangle$

The value of an *expression* statement is simply the value of the expression, and the value of a *sequence* statement is the value of evaluating the second statement, using the environment from the result of evaluating the first statement.

Functions

SIMPL supports top-level functions that must be defined at the beginning of the program. A function declaration$\langle F \rangle$ consists of an identifier followed by a parameter list with zero or more identifiers and finally a function body which is one or more statements.

$\langle F \rangle ::= \langle Id \rangle\ (\langle Id \rangle^*)\ \{\ \langle E \rangle\ \}$

A function call then consists of an identifier, referencing a function declaration, followed by a list of expressions which is the function arguments:

$\langle E \rangle ::= \langle Id \rangle\ (\langle E \rangle^*)$

The length of the argument list and the parameter list in the declaration must be equal. Furthermore the expressions in the argument list must evaluate to either integers or boolean values. The value of a function call is the value of the final statement evaluated in the function body. Since expressions only return values, functions does not have any side effects.

Programs

We finally define the syntax of a *SIMPL* program, which is one or more function declarations, followed by a function call.

$\langle P \rangle ::= \langle F \rangle\ \langle F \rangle^*\ \langle Id \rangle\ (\langle E \rangle^*)$

5.2 Concrete interpreter for *SIMPL*

To interpret the language, we have implemented an interpreter in the programming language *Scala*. The implementation is purely functional, so we avoid any state and side-effects. We translate the grammar we just described into the following object:

```
1   object ConcreteGrammar {
2     sealed trait ConcreteValue
3     object ConcreteValue {\\
4       case class True() extends ConcreteValue
5       case class False() extends ConcreteValue
6       case class IntValue(v: Int) extends ConcreteValue
7       case class UnitValue() extends ConcreteValue
8     }
9     sealed trait Exp
10    sealed trait Stm
11    case class Id(s: String)
12    case class FDecl(name: Id, params: List[Id], stm: Stm)
13    case class Prog(funcs: HashMap[String, FDecl], fCall: CallExp)
14  }
```

Figure 5.1: High level overview of grammar implementation in scala.

The interpreter consists of the following functions:

```
1   class ConcreteInterpreter {
2     def interpProg(p: Prog): Result[ConcreteValue, String]
3
4     def interpExp(p: Prog,
5             e: Exp,
6             env: HashMap[Id, ConcreteValue]): Result[ConcreteValue, String]
7
8     def interpStm(p: Prog,
9             s: Stm,
10            env: HashMap[Id, ConcreteValue]):
11            Result[(ConcreteValue, HashMap[Id, ConcreteValue]), String]
12  }
```

Figure 5.2: High level overview of the interpreter implementation in Scala. For the full implementation, see apendix A.

We use an immutable map of type *HashMap[Id, ConcreteValue]* to represent our program environment.

The interpretation is started by a call to *interpProg* with a program *p*, which then call calls *interpExp(p, p.fCall, env)* where *env* is a fresh environment. This means that the function referenced by *p.fCall* acts as a main-function.

5.2.1 Error handling

We wish to keep our implementation purely functional, so we need to avoid throwing exceptions whenever we encounter an error. Instead we define a trait *Result[+V, +E]*, that can either be *Ok(v: V)*, or *Error(e: E)*, for some types *V* and *E*.

```
1   trait Result[+V, +E]
2   case class Ok[V](v: V) extends Result[V, Nothing]
3   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 and flatMap

We define two functions, *map* and *flatMap* for *Result*:

```
1  def map[T](f: V => T): Result[T, E] = this match {
2  case Ok(v) => Ok(f(v))
3  case Error(e) => Error(e)
4  }
5
6  def flatMap[EE >: E, T](f: V => Result[T, EE]): Result[T, EE] = this match {
7  case Ok(v) => f(v)
8  case Error(e) => Error(e)
9  }
```

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 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.

Chapter 6

Symbolic execution of *SIMPL*

In this chapter we will describe a symbolic interpreter for *SIMPL* .

6.1 Extension of grammar

In order to symbolically interpret *SIMPL* , we must extend the grammar for the language, to include symbolic values. A symbolic value can either be a symbolic integer, a symbolic boolean or the unit value.

$\langle SV \rangle$ ::= $\langle SI \rangle$ | $\langle SB \rangle$ | unit

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

$\langle I \rangle$ ::= 0 | 1 | -1 | 2 | -2 | ...

$\langle Sym \rangle$ ::= a | b | c | ...

$\langle SI \rangle$ ::= $\langle I \rangle$
| $\langle Sym \rangle$
| $\langle SI \rangle$ + $\langle SI \rangle$ | $\langle SI \rangle$ - $\langle SI \rangle$ | $\langle SI \rangle$ * $\langle SI \rangle$ | $\langle SI \rangle$ / $\langle SI \rangle$

A symbolic boolean can either be *True*, *False* or 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.

$\langle B \rangle$ ::= True | False

$\langle SB \rangle$ :: = $\langle B \rangle$
| $\langle SI \rangle$ <$\langle SI \rangle$ | $\langle SI \rangle$ >$\langle SI \rangle$ | $\langle SI \rangle \leq \langle SI \rangle$ | $\langle SI \rangle \geq \langle SI \rangle$ | $\langle SI \rangle$ == $\langle SI \rangle$
| ! $\langle SB \rangle$

Note that the definition of symbolic values also contain the concrete values, so we change the grammar of expressions to include symbolic values instead of just concrete values.

$\langle E \rangle ::= \langle SV \rangle$

6.2 Path-constraints

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

```
case class PathConstraint(conds: List[SymbolicBool],
  ps: PathStatus)
  sealed trait PathResult
  case class Certain() extends PathResult
  case class Unknown() extends PathResult
```

This definition consists of two elements. *conds* is the list of conditions that must be met to follow the given execution path. 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*.

6.3 Interpretation of expressions

To interpret an expression, we define the following function

```
def interpExp(p: Prog,
  e: Exp,
  env: HashMap[Id, SymbolicValue],
  pc: PathConstraint): List[ExpRes]

case class ExpRes ExpRes(res: Result[SymbolicValue, String],
  pc: PathConstraint)
```

This definition is similar to the function from the concrete interpreter, except that the function now also takes a *path-constraint*as argument. The return type has also changed so that it now includes a *path-constraint*aswell. Further we return a list of such pairs, since expression can have several possible values depending on which execution path is followed.

6.3.1 Arithmetic and boolean 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 now 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 integers and compute the result.

```
for {
  r1 <- interpExp(p, e1, env, pc)
  r2 <- interpExp(p, e2, env, r1.pc)
} yield ExpRes(
  r1.res.flatMap(v1 => r2.res.flatMap(
  v2 => (v1, v2) match {
    case (i: IntValue, j: IntValue) => evalInts(i.v, j.v, op)
    case (i: SymbolicInt, j: SymbolicInt) => Ok(SymbolicAExp(i, j, op))
    case _ => Error("arithmetic operations on non integer values")
  })),
  r2.p
)
```

Boolean expressions are interpreted in a similar fashion.

6.3.2 Function calls

Consider a Call expression *CallExp(id: Id, args: List[Exp])*. To interpret this, we must first check that the function is defined and if not, we immediately return an *Error*. Otherwise, we check that the formal and actual argument list does not differ in length. Finally, if both of these things check out, we must interpret the expressions given in the *args* list, construct a local environment with the argument and interpret the statement in the function body. Given an argument list $[e_1, e_2, \ldots, e_n]$, we map *interpExp* on to the list, which gives a list of lists $[L_{e_1}, L_{e_2}, \ldots, L_{e_n}]$. Each list contains the possible values for the given argument, so from this we must construct all possible argument lists, which is the cartesian product over all n lists $L_{e_1} \times L_{e_2} \times \ldots \times L_{e_n}$. For each possible argument list we attempt to build a local environment. If encounter an error during this process, either from the interpretation of the arguments, or from an argument having a unit value, this error will be the of function call with the given argument list. Otherwise the result will be the interpretation of the function body with the local environment.

Chapter 7

Concolic execution of *SIMPL*

Chapter 8

Conclusion

Appendix A

Source code

Appendix B

Figures

Bibliography

[1] 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.