Symbolic Execution(Working title)

Aarhus Universitet

Søren Baadsgaard

May 28, 2019

Abstract

Contents

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.

```
procedure ComputeRevenue(units, cost)
    revenue := 2 · units
    if revenue ≥ 16 then
        revenue := revenue − 10
        assert revenue ≥ cost
    end if
    return revenue
end procedure
```

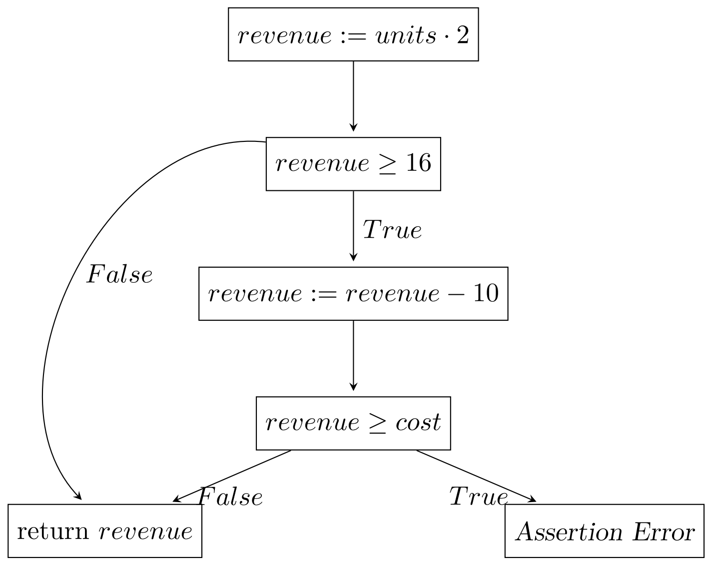

Figure 2.1: Control-flow graph for ComputeRevenue

After applying the discount, we assert that $revenue \geq cost$ 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] [4].
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 illustrate the relationship between concrete and symbolic values:

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

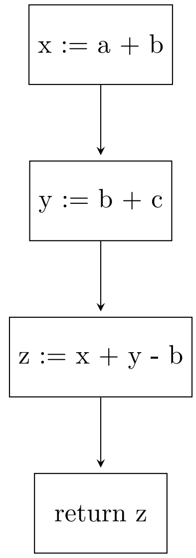

Figure 3.1: Control-flow graph for ComputeSum

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 $s_1, s_2, \dots, s_n$ with conditions $q_1, q_2, \dots, q_n$, 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 $[q_1, q_2, \dots, q_k]$ 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 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 select $\alpha = 10$ 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:

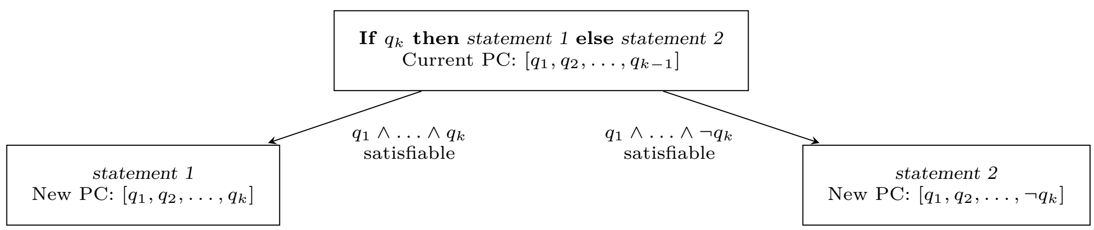

Figure 3.2: Abstract overview of the symbolic execution of an *if*-statement, which potentially leads to two new execution paths, each with a new *path-constraint* .

- **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 [4].

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

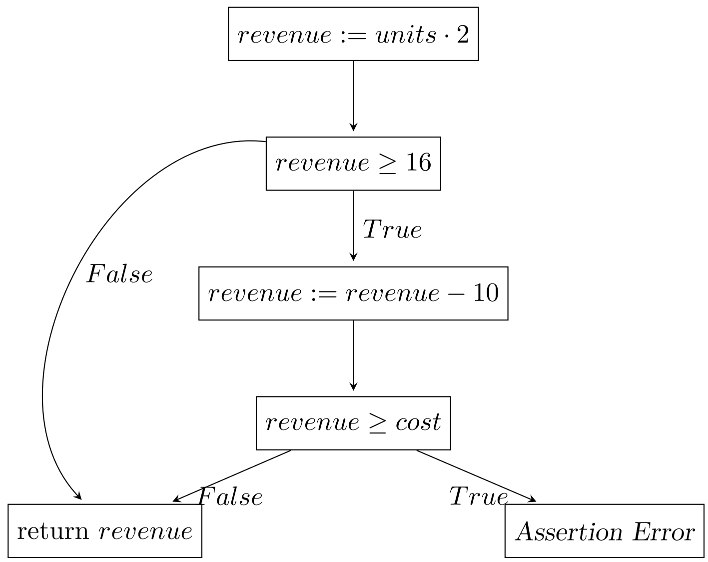

Figure 3.3: Control-flow graph for ComputeRevenue

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 an *if*-statement with condition $\alpha \cdot 2 \geq 16$. 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 path corresponding to the condition evaluating to *true*. We also start a new execution with the same environment and a *path-constraint* equal to $[true, \neg(\alpha \cdot 2 \geq 16)]$. 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 *revenue* and then reach an *assert*-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 with 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. $true \wedge (\alpha \cdot 2 \geq 16) \wedge \neg(((2 \cdot \alpha) - 10) \geq \beta)$.

The first two *path-constraints* corresponds to the two different paths that leads to the return ment, 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* 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.

3.3 Constraint solving

As we just described, a symbolic execution of a program results in one or more *path-constraints* corresponding to each possible execution path. We know that each of these *path-constraints* are satisfiable, so we can solve the *path-constraint* by finding an assignment of concrete values to the symbols, that causes it to evaluate to true. Consider for example

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

which is one of the three resulting *path-constraints* from the motivating example. From the first condition we get that $\alpha \geq 8$, so we can select $\alpha = 8$. The second condition then gives us that $6 \geq \beta$, so we can select $\beta = 6$. If we do a concrete execution of the program with $units = 8$ and $cost = 6$, it will follow the execution path that correspond to this *path-constraint* . We can do the same for the remaining two *path-constraints*, and in the end we will have a pair of concrete input values for each possible execution path. So symbolic execution not only allows us to explore all possible execution paths, it also allows to generate a small set of concrete input values that cover all these paths.

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 expressions. 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. In this case, the number of execution paths is potentially infinite [1]. 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 $i \leq b$ **do**
    $r := r \cdot a$
    $i := i + 1$
  end while
  return r
end procedure

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

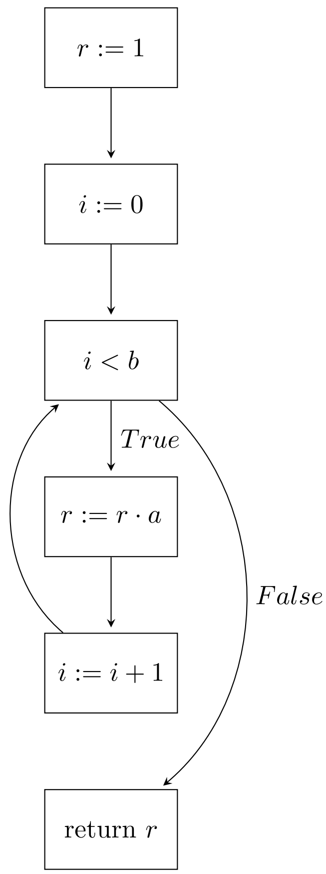

Figure 3.4: Control-flow graph for ComputePow

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. 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.

3.4.2 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* to 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)* problem. To understand SMT, we first consider the *Boolean Satisfiability(SAT)* problem. In this problem we a given a logical formula over boolean variables $q_1, q_2, \ldots, q_n$. We want to decide if there exists an assignment of truth values to each variable such that the formula evaluates to true. for example, $(q_1 \wedge \neg q_2)$ is a *yes*-instance of this problem, since $q_1 = true$ and $q_2 = false$ causes the formula to evaluate to *true*. On the other hand $q_1 \wedge q_2 \wedge \neg q_2$ is a *no*-instance since the formula is clearly false no matter which values we assign. This problem is *decidable*, meaning that we can always answer *yes* or *no* given a formula. However it is also NP-complete, which means that the best known method for solving this problem takes worst-case exponential time. SMT is then an extention of this problem. We now let the boolean variables $q_1, q_2, \ldots, q_n$ represent expressions from some theory such as the *theory of integer linear* *arithmetic(LIA)*. In LIA, an expression e is defined as

$$
\begin{array}{lcl}
e \in Expression & := & p \bowtie p \\
\bowtie \in Comparison & := & \leq \ | \ \geq \ | \ = \\
p \in Polynomial & := & a \mid a \cdot x \mid p \circ p \\
\circ \in Operator & := & + \mid - \\
a \in Coefficient & := & \mathbb{Z} \\
x \in Variable & := & \mathbb{Z}
\end{array}
$$

We now want to decide if the formula is satisfiable with respect to the theory.

The theory of linear integer arithmetic

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

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

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. We can check the feasibility of the **ILP** using the *branch-and-bound* algorithm. If it is feasible, we will also get a satisfying assignment.

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 environment maps variables to *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 [3]. 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 Maintaining the environments

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 Generating input values for 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 a new set of input values, we make a new *path-constraint* by negating the final condition in the original *path-constraint* . 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 are not constrained by the *path-constraint* we keep the current value for the next run.

To illustrate concolic execution, we consider the program from the motivating example:
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. We get a new *path-constraint* which is $[(2{\cdot}\alpha \geq 16)]$. Next, we reach another *assign* statement, so we get the following environments:

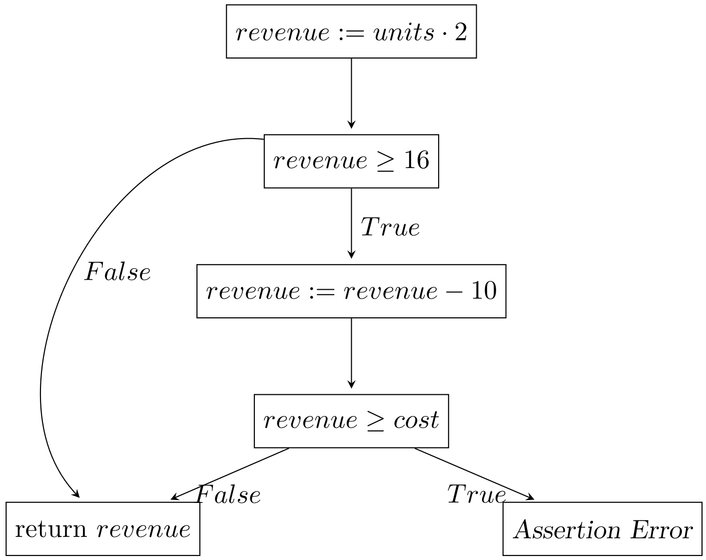

Figure 4.1: Control-flow graph for ComputeRevenue

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

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*

$$
\begin{aligned}
2 \cdot \alpha &\geq 16 \\
(2 \cdot \alpha - 10) &< \beta.
\end{aligned}
$$

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. 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 re execute the program with $units = 5$ and $cost = 7$. 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 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 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 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 state, 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[3]. 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. This allows us to still explore more execution paths, without potentially exploring infeasible paths.

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 | ...

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

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:

$\langle S \rangle ::= \text{if } \langle E \rangle \text{ then } \langle S \rangle \text{ else } \langle S \rangle$
$\quad | \quad \text{while } \langle E \rangle \text{ do } \langle S \rangle$
$\quad | \quad \text{assert } \langle E \rangle$

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:

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

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:

```
object ConcreteGrammar {
  sealed trait ConcreteValue
  object ConcreteValue {\\
    case class True() extends ConcreteValue
    case class False() extends ConcreteValue
    case class IntValue(v: Int) extends ConcreteValue
    case class UnitValue() extends ConcreteValue
  }
  sealed trait Exp
  sealed trait Stm
  case class Id(s: String)
  case class FDecl(name: Id, params: List[Id], stm: Stm)
  case class Prog(funcs: HashMap[String, FDecl], fCall: CallExp)
}
```

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

The interpreter consists of the following functions:

```scala
class ConcreteInterpreter {
  def interpProg(p: Prog): Result[ConcreteValue, String]

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

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

Figure 5.2: High level overview of the interpreter implementation in *Scala*.

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*.

```scala
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 and flatMap

We define three functions *map* and *flatMap* and traverse:

```
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 $f : V \rightarrow T$ 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 $f : V \rightarrow Result[W, E]$ 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* . We start by extending the grammar to include symbolic values. Next, we describe the implementation of *path-constraints* and the functions for interpreting expressions and statements.

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 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.

$\langle B \rangle$ ::= 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) {
    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*. For *PathConstraint*, we also define a method *:+* which takes a symbolic boolean, and returns a new *path-constraint* with the boolean added to *conds*.

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,
  forks: Int): 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 takes two extra parameters *pc*, which is the current *path-constraint* and *forks* which is the current number of forks. The return type is now a list of pairs *(Result[SymbolicValue, String], PathConstraint)*, with a pair for each possible execution path.

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 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. Boolean expressions are interpreted in a similar fashion.

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 Interpreting statements

To interpret statements, we define the following function:

```
interpStm(p: Prog,
    stm: Stm,
    env: HashMap[Id, SymbolicValue],
    pc: PathConstraint,
    forks: Int): List[StmRes]
```

```
case class StmRes(res: Result[SymbolicValue, String],
                  env: HashMap[Id, SymbolicValue],
                  pc: PathConstraint)
```

The signature is similar to *interpExp*, except that the return type now also includes a possibly updated environment.

6.4.1 Assignment statements

Given an *Assignment*-statement *AssignStm(v: Var, e: Exp)*, we interpret the expression e, and get a list of results for each possible execution path. For each of these results, we return the value of the expression and an updated environment. If the expression resulted in an error, or a unit value, we instead return error and the original environment.

6.4.2 Conditional statements

We define the following classes to represent each type of conditional statement:

```
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()*, we follow the appropriate execution path, with the original values of *pc* and *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.

6.4.3 Sequence statements

We define the following class to represent a sequence statement:

```
case class SeqStm(s1: Stm, s2: Stm)
```

For each possible result from interpreting *s1*, we interpret *s2* with the new environment. If the interpretation of *s1* results in an error, this will be the result of the sequence expression. otherwise it will be the result of the interpretation of *s2*.

6.4.4 Expression statements

We define the following class to represent an *expression*-statement:

```
case class ExpStm(e: Exp)
```

To interpret this, we interpret e. For each result of this, the result of the statement, will be the value of the expression and the original environment.

Chapter 7

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.

[2] 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`.

[3] 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`.

[4] 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`.