Symbolic Execution(Working title)

Aarhus Universitet

Søren Baadsgaard

April 23, 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
```

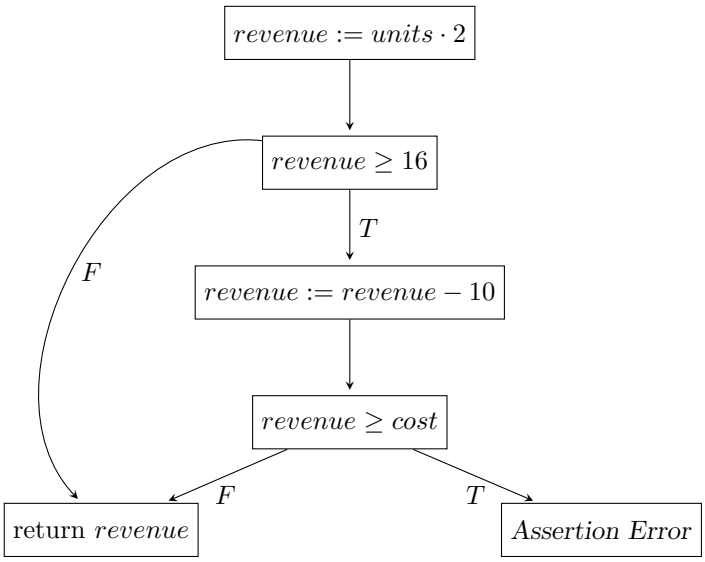

Figure 2.1: Control-flow graph for ComputeRevenue

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

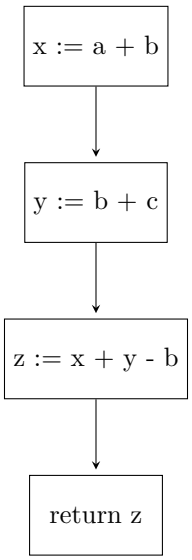

Figure 3.1: Control-flow graph for ComputeSum

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*:

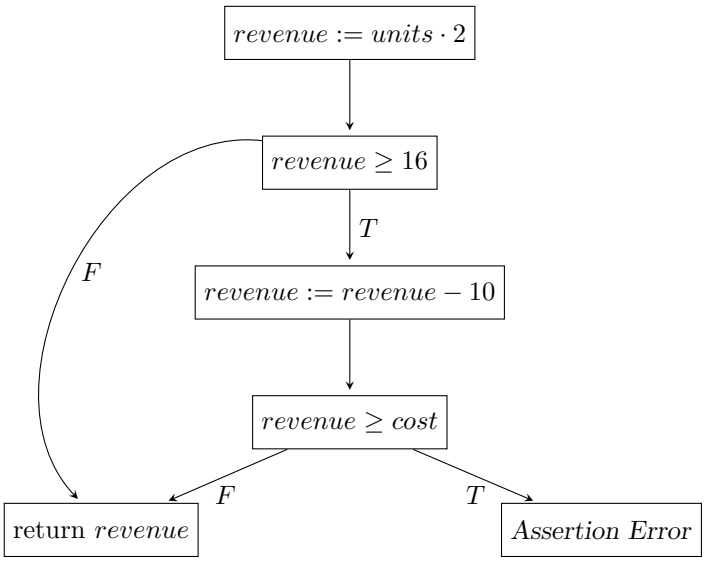

Figure 3.2: Control-flow graph for ComputeRevenue

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
```

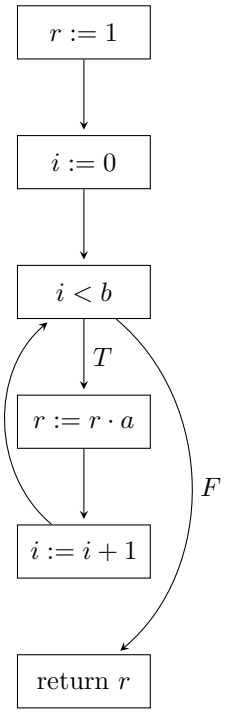

Figure 3.3: Control-flow graph for ComputePow

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 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

Chapter 5

Introducing the language *EXP*

In this chapter we will introduce *EXP* which is a small programming language which consists of top-level functions and expressions.

5.1 Syntax of *EXP*

The main building block of *EXP* are expressions which we characters as follows:

Basic expressions

The basic expressions consists of integers$\langle I, \rangle$, booleans$\langle B \rangle$, and variables which we reference by identifiers$\langle Id \rangle$. Furthermore they consist of arithmetic operations and comparisons of integers. Finally an expression can be one expression, followed by another.

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

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

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

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

Variable declaration and assignment

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

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

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

Conditional expressions

EXP supports two different conditional expressions, namely *if-then-else* expressions and *while* expressions:

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

The condition expression must evaluate to a boolean value for both these expressions. The value of an *if-then-else* expression is the value of the expression that ends up being evaluated, depending on the condition. In a *while* expression, we are not guaranteed that the second expression is evaluated, so we introduce a special *unit*-value which will be the value of any *while*-expression.

Functions

EXP supports toplevel 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 an expression.

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

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 expression in the function body.

Programs

We finally define the syntax of a *EXP* program, which is zero or more function declarations, followed by a root expression.

$\langle P \rangle ::= \langle F \rangle^* \langle E \rangle$

Chapter 6

Symbolic execution of EXP

Chapter 7

Concolic execution of EXP

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.