

Symbolic Execution(Working title)

Aarhus Universitet



Søren Baadsgaard

February 7, 2019

Abstract

Contents

1	Introduction	2
2	Summary of theory	3
2.1	Principles of symbolic execution	3
2.1.1	symbolically executing a program	4
2.1.2	Handling branching in a program	5
2.1.3	Symbolic execution trees	6
3	Basic symbolic execution for the <i>SImPL</i> language	7
3.1	description	7
3.2	Introducing the <i>SImPL</i> language	7
3.2.1	Interpreting <i>SImPL</i>	8
4	Further extensions	9
5	Conclusion	10
A	Source code	11
B	Figures	12

Chapter 1

Introduction

Chapter 2

Summary of theory

In this chapter we will cover some of the theory behind symbolic execution. We will start by describing what it means to *symbolically execute* a program and how we deal with potential branching. We will also describe the connection between a symbolic execution of a program, and a concrete one. Furthermore we will relate this technique to alternatives such as formally proving correctness.

2.1 Principles of symbolic execution

in [1], the author looks at a spectrum of ensuring that a program functions as expected. In one extreme we have program testing, where the programmer specifies a sample of input. The program is then run on this sample, and if the program runs as intended, the programmer can be sure that for this sample the program is correct. Choosing a proper sample of inputs will then enable the programmer to have some level of confidence that the program runs correctly. But since most programs can take virtually an infinite number of different inputs, one can never be absolutely sure that the program is free of bugs. At the other end of the spectrum is formal program verification. This requires a proper mathematical specification of the program, and a proof procedure that will verify that this specification is correct w.r.t some formal requirements. This will give the programmer a complete confidence that the program is in fact correct. This comes at the cost of producing a proper and adequately strong specification, performing sound steps in the proof procedure as well as having a strong set of requirements. We find symbolic execution somewhere in the middle of this spectrum. While we do not build a formal specification and apply any sort of program verification, we do not restrict ourself to specific samples of input. Instead, we try to test the program for whole classes of inputs (such as all integers that are a power of two), which is represented by symbolic values instead of actual ones.

2.1.1 symbolically executing a program

Symbolic execution is described in [1] by imagining a simple programming language which restricts all variables to signed integers. The language supports the usual arithmetic operations on these integers, as well as a conditional expression to decide whether a given value is ≤ 0 . To execute such a program symbolically, we allow the variables to also take *symbolic* values, where these symbols represent some signed integer. We also allow arithmetic operations on these symbols, and for variables to hold such expressions. This means variables in fact contains polynomials over the integers. To illustrate this, we consider the following simple program, that takes parameters a, b, c and computes the sum:

```
Fun sum(a, b, c) {  
  var x = a + b  
  var y = b + c  
  var z = x + y - b  
  return z  
}
```

If we run this program on concrete inputs, say $a = 2, b = 3, c = 4$, we would get the following execution:

1. $x = 2 + 3 \Rightarrow y = 5$
2. $y = 3 + 4 \Rightarrow y = 7$
3. $z = 5 + 7 - 3 \Rightarrow z = 9$
4. return 9

So we see that on the specific input $a = 2, b = 3, c = 4$, the program returns the correct value.

Let us now run the program with symbols. Say that we the input the following symbolic values: $a = \alpha, b = \beta, c = \gamma$. The execution would then look like:

1. $x = \alpha + \beta$
2. $y = \beta + \gamma$
3. $z = (\alpha + \beta) + (\beta + \gamma) - \beta$
4. return $\alpha + \beta + \gamma$

From this execution we can actually conclude that the program will return the correct result for any three integers that we give as input.

2.1.2 Handling branching in a program

In the previous section we gave an example of a symbolic execution of a simple program that computes the sum of three integers. The program is an extremely simple case, in which the program will behave exactly the same for any possible input values. In reality however, most program languages have some way of allowing branching to happen, and with this feature, we cannot guarantee that the program will behave exactly the same for all inputs. If our program contains an expression like *If $x > 2$ Then ... Else ...*, the program will behave differently depending on whether or not $x > 2$. To encapsulate this, we introduce a *path-constraint* (PC), which is a boolean expression that will contain all properties that the input must satisfy to follow the given path. At the beginning of the execution, the PC will be initialized with the value *true* as no assumptions have been made yet (If we introduce pre-conditions on the input, the PC will of course contain these). At any *if-expression* with condition q , during the execution we will look at the following to expressions

1. $PC \supset q$
2. $PC \supset \neg q$

where expression 1 tells us that q is contained in the PC , and expression 2 tells us that the opposite of q is contained in the PC . It is clear that at most one of these expressions can be true at any given time, which leaves us with three possible outcomes. If expression 1 holds, we simply follow the then-branch, and if expression 2 holds, we follow the else branch. In both these outcomes, PC is not updated since it already contains the property that leads to the given branch that was chosen. The third outcome happens when neither expression holds. At this point we cannot simply follow one of the branches, so we must fork our execution into two parallel executions, one where we assume that q holds and one where we assume $\neg q$ holds. This will produce two new PC s, namely

$$\begin{aligned} PC' &\leftarrow PC \wedge q \\ PC'' &\leftarrow PC \wedge \neg q \end{aligned}$$

and from here we will have two separate executions, one with PC' that will follow the then-branch, and one with PC'' that will follow the else-branch. It is important to note that the PC can never become identically *false*. To see this we first note that PC will always start with the value *true*, and all other updates will be of the form $PC \leftarrow PC \wedge r$ where $r \in \{q, \neg q\}$. But these updates only happen whenever we cannot infer q or $\neg q$ from the existing value of PC and it will only receive exactly of q or $\neg q$. So it is never possible to have a state of PC in which we have $\dots \wedge q \wedge \dots \wedge \neg q \dots$.

As in [1], we can illustrate this with the following program that computes a^b .

```

Fun pow(a, b) {
  var r = 1
  var i = 0
  while (i < b) {
    r = r*a
    i = i + 1
  }
  return r
}

```

If we assign $a = \alpha$ and $b = \beta$, we get the following execution:

- PC is initialized to *true*
- $r \leftarrow 1$
- $i \leftarrow 0$
- We hit a branching point, so we check if $true \supset (0 < \beta)$ or $true \supset \neg(0 < \beta)$. Since neither of these hold, we must fork:
 - **Case** $\neg(0 < \beta)$: $PC \leftarrow true \wedge \neg(0 < \beta)$. The program returns 1. So we can conclude that the program returns 1 when $\beta \leq 0$.
 - **Case** $(0 < \beta)$: $PC \leftarrow true \wedge (0 < \beta)$.
 - $r \leftarrow 1 \cdot a$
 - $i \leftarrow 0 + 1$
- We hit a branching point again, so we check if $true \wedge (0 < \beta) \supset 1 < \beta$ or $true \wedge (0 < \beta) \supset \neg(1 < \beta)$. Since neither of these holds, we fork again:
 - **Case** $\neg(1 < \beta)$: $PC \leftarrow true \wedge (0 < \beta) \wedge \neg(1 < \beta)$. The program returns α . So we can conclude that the program returns α when $\beta = 1$.
 - **Case** $1 < \beta$: $PC \leftarrow true \wedge (0 < \beta) \wedge (1 < \beta)$.
 - $r \leftarrow a * a$
 - $i \leftarrow 1 + 1$
- We hit a branching point ...

2.1.3 Symbolic execution trees

Chapter 3

Basic symbolic execution for the *SImPL* language

3.1 description

In this chapter we will describe the process of implementing symbolic execution for a simple imperative language called *SImPL*.

3.2 Introducing the *SImPL* language

SImPL (Simple Imperative Programming Language) is a small imperative programming language, designed to highlight the interesting use cases of symbolic execution. The language supports only one type, namely the set integers \mathbb{N} . Furthermore we will interpret 0 as *false* and any other integer as *true*. *SImPL* supports basic variables that can be assigned the value of any expression, as well as basic branching functionality through an **If - Then - Else** statement. Furthermore it allows for looping through a **While - Do** statement.

We will describe the language formally, by the following Context Free Grammar:

$$\langle int \rangle ::= 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \dots$$

$$\langle var \rangle ::= a \mid b \mid c \mid \dots$$

$$\begin{aligned} \langle exp \rangle ::= & \langle int \rangle \\ & \mid \langle var \rangle \\ & \mid \langle exp \rangle + \langle exp \rangle \mid \langle exp \rangle - \langle exp \rangle \mid \langle exp \rangle * \langle exp \rangle \mid \langle exp \rangle / \langle exp \rangle \\ & \mid \langle exp \rangle > \langle exp \rangle \mid \langle exp \rangle == \langle exp \rangle \end{aligned}$$

$$\begin{aligned} \langle stm \rangle ::= & \langle exp \rangle \\ & \mid \langle var \rangle = \langle exp \rangle \\ & \mid \langle stm \rangle \langle stm \rangle \\ & \mid \text{if } \langle exp \rangle \text{ then } \langle stm \rangle \text{ else } \langle stm \rangle \\ & \mid \text{while } \langle exp \rangle \text{ do } \langle stm \rangle \\ & \mid \text{Print E} \end{aligned}$$

$$\langle prog \rangle ::= \langle stm \rangle$$

where $+$, $*$, $-$, $/$ denotes the usual arithmetic operators on integers, and $>$, $==$ denotes the comparison-operators of *greater-than* and *equal-to* respectively. When interpreting a comparison-operator we will return 0 for *false* and 1 for *true*. Finally we see that a program $\langle prog \rangle$ is simply a sequence of one or more statements.

3.2.1 Interpreting *SImPL*

In order to work with *SImPL*, we have build a simple interpreter using the *Scala* programming language. We use a simple recursive strategy, where we recursively interpret statements from left to right, each time returning a potentially updated *environment* containing bindings of variable names to integer values.

Chapter 4

Further extensions

Chapter 5

Conclusion

Appendix A

Source code

Appendix B

Figures

Bibliography

- [1] 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>.