Symbolic Execution(Working title)

Aarhus Universitet

Søren Baadsgaard

February 12, 2019

Abstract

Contents

Chapter 1

Introduction

Chapter 2

Summary of theory

In this chapter we will cover some of 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.

2.1 Principles of symbolic execution

In [2], symbolic execution is described as a practical approach between simple program testing and program proving. At one extreme, program testing allows the programmer to get some level of confidence in the program, by running it on a well-selected sample of input values, but this sample size will be but a fraction of the possible input values. At the other extreme, program proving will give complete confidence in the programs correctness. To achieve this, one must provide a precise specification of correct behavior, as well as be able to perform perform formal proof steps to conclude that the program satisfies this specification. This is a challenging task, even for relatively simple programs. *Symbolic execution* will serve as a practical middle ground between these two, in which we try to extend simple program testing to cover more general classes of inputs.

2.1.1 symbolically executing a program

[1] describes symbolic execution as follows:

Any input to the program will be replaced with *symbolic* values instead of concrete ones. Any operations on the symbolic values, will result in expressions over these, so our program state can be describes as a map from variable names to expressions over the input values. This also means that any return value from the execution will be such an expression. In [2] an example is given of a program written in a simple language with only signed integer values and

arithmetic operations on these. In a symbolic context, this will translate to a program that operates on polynomials with integer coefficients.

To illustrate this, we consider the following simple program, that takes parameters a, b, c and computes the sum:

```
1   Fun sum(a, b, c)
2     var x = a + b
3     var y = b + c
4     var z = x + y - b
5     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 ⇒ y = 5
2. y = 3 + 4 ⇒ y = 7
3. z = 5 + 7 - 3 ⇒ 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 nor $x > 2$. 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. $PC \wedge q$
2. $PC \wedge \neg q$.

This gives a number of possible scenarios:

- **Only the first expression is satisfiable**: We will update PC with q so $PC \leftarrow PC \wedge q$. Execution will continue by following the *then*-branch.
- **If only the second expression is satisfiable**: We will update PC with $\neg q$ so $PC \leftarrow PC \wedge \neg q$. Execution will continue by following the *else*-branch.
- **Both expressions are satisfiable**: In this case, the execution can follow both branches, so we *fork* the program, by updating PC with $PC \leftarrow q$, and we make a copy of PC and update this with $PC' gets \neg q$. This gives us two executions, one that follows the *then*-branch with PC and one that follows the *else*-branch with PC'.

We illustrate this with the following example:

```
1  Fun pow(a, b)
2    var r = 1
3    var i = 0
4    while (i < b)
5      r = r*a
6      i = i + 1
7    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 $true \wedge (0 < \beta)$ and $true \wedge \neg(0 < \beta)$ are satisfiable. Since they both are, 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 $true \wedge (0 < \beta) \wedge (1 < \beta)$ and $true \wedge (0 < \beta) \wedge \neg(1 < \beta)$ are satisfiable. Since both they are, 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** $1 < \beta$: $PC \leftarrow true \wedge (0 < \beta) \wedge (1 < \beta)$.

  $r \leftarrow a * a$

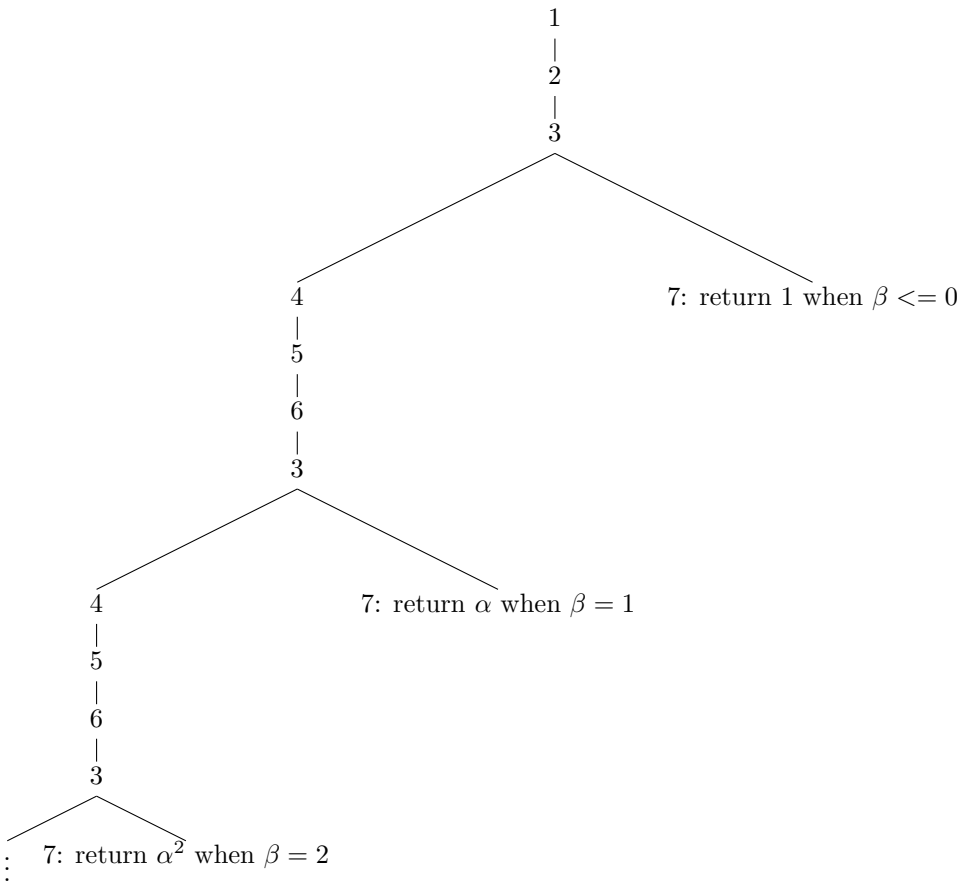
  $i \leftarrow 1 + 1$
- We hit a branching point ...

An important property of the *path-constraint* is that it can never become identically false. To see why this is the case, we have to look at the possible updates of PC. At the start of an execution, it will be initialized with the value *true*. At any branching point, it will be updated with exactly one of the expressions $PC \wedge q$ and $PC \wedge \neg q$, and only if the given expression is satisfiable. So PC will never end up looking like $\ldots \wedge q \wedge \ldots \wedge \neg q \wedge \ldots$ for some condition q. What this means is that when the program terminates at the end of some execution path, PC will be a satisfiable formula over the symbolic values, which means that we can solve the constraints and derive a set of concrete values which will follow the exact same path if we execute the program normally.

2.1.3 Limitations of symbolic execution

Infinite execution trees

As demonstrated in the last example in the previous sections, a symbolic execution can easily become infinite as soon as we introduce branching and some looping structure. This is further illustrated if we consider an execution tree for a program. In [2], they are described by enumerating each statement, and let each node in the tree, correspond to an execution of one of the command. The edges going out from a node corresponds to the transition from one statement to the next. From this we can see that non forking statements will only have a single edge outgoing, while forking statements will have two. The forking edges will then correspond to splitting up the execution and following a different path, with a different *path-constraint*. As an example, we can look at the execution tree for the program *pow*, that computes a^b.



As we can see, this tree is infinite, since $b = \beta$ and β can be arbitrarily large. In this case, our symbolic execution would run forever if we insisted on exhaustively exploring all possible paths. This illustrates one of the limitations of symbolic execution. For programs with infinite execution trees, we simply cannot exhaust all possible inputs, so we have to restrict our testing to exploring only a finite number of paths.

(Maybe write something about induction over trees, and finite trees allowing for exhaustive search)

the ability(or inability) to decide whether a given path is feasible

(Write something about restrictions on SMT-solvers e.g SAT being NP Complete and some theories may be undecidable)

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 two types, namely the set integers $\mathbb{N}$ and the boolean values $true$ and $false$. ***SImPL*** supports basic variables that can be assigned integer values, as well as basic branching functionality through an **If - Then - Else** statement. Furthermore it allows for looping through a **While - Do** statement. It also supports top-level functions and the use of recursion.

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

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

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

$\langle exp \rangle$ ::= $\langle aexp \rangle$ | $\langle bexp \rangle$ | $\langle nil \rangle$

$\langle nil \rangle$ ::= ()

$\langle bexp \rangle$ ::= True | False
| $\langle aexp \rangle > \langle aexp \rangle$
| $\langle aexp \rangle == \langle aexp \rangle$

$\langle aexp \rangle$ ::= $\langle int \rangle$ | $\langle id \rangle$
| $\langle aexp \rangle + \langle aexp \rangle$ | $\langle aexp \rangle$ - $\langle aexp \rangle$
| $\langle aexp \rangle \cdot \langle aexp \rangle$ | $\langle aexp \rangle$ / $\langle aexp \rangle$
| $\langle cexp \rangle$

$\langle cexp \rangle$ ::= $\langle Id \rangle$ ($\langle aexp \rangle$*) # Call expression

$\langle stm \rangle$ ::= $\langle exp \rangle$
| $\langle Id \rangle = \langle aexp \rangle$
| $\langle stm \rangle$ $\langle stm \rangle$
| if $\langle exp \rangle$ then $\langle stm \rangle$ else $\langle stm \rangle$
| while $\langle exp \rangle$ do $\langle stm \rangle$

$\langle fdecl \rangle$ ::= $\langle Id \rangle$ ($\langle Id \rangle$*) $\langle fbody \rangle$

$\langle fbody \rangle$ ::= $\langle stm \rangle$
| $\langle fdecl \rangle$ $\langle fbody \rangle$

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

Expressions

expressions comes in three different types, arithmetic, boolean and a *nil* expressions.

arithmetic expressions consists of integers, variables referencing integers, or the usual binary operations on these two. We also consider function calls an arithmetic expression, but assigning anything other than integers to a variable will result in a runtime error. **boolean expressions** consists of the boolean values $true$ and $false$, as well as comparisons of arithmetic expressions.

nil nil expression is a special case which is only returned from *while* statements.

Statements

Statements consists of assigning integer values to variables, *if-then-else* statements for branching and a *while-do* statement for looping. Finally a statement can simply an expression, as well as a compound statement to allow for more than one statement to be executed.

Function declarations

Function declarations consists of an identifier, followed by a list of zero or more identifiers for parameters, and finally a function body which is simply a statement. Functions does not have any side effects, so any variables declared in the function body will be considered local. Furthermore, any mutations of globally defined variables will only exist in the scope of that particular function.

programs

We consider a program to be zero of more top-level function declarations, as well a statement. The statement will act as the starting point when executing a a program.

3.2.1 Interpreting *SImPL*

In order to work with ***SImPL*** , we have build a simple interpreter using the *Scala* programming language.

We represent our environment as a map

$$env : \langle Id \rangle \to \mathbb{N} \tag{3.1}$$

and each time we interpret a statement we return an instance of this map that reflects the current state of the environment. Note that we restrict both variables and function arguments to integer values.

A program is represented as an object $Prog$ which carries a map

$$funcs : \langle Id \rangle \to \langle fdecl \rangle$$

of top-level function declarations, as well as a root statement. To interpret the program we simply traverse the tree starting with the root statement. Interpreting function calls consists of looking up the function in the $Prog$ object, add the function parameters to the environment and then interpreting the statement in the function body. In order to keep functions from having side effects, we simply return the original environment as it was before adding function parameters and interpreting the body.

3.2.2 Symbolic interpreter for *SImPL*

To be able to do symbolic execution of a program written in ***SImPL*** , we must extend our implementation to allow for symbolic values to exists. To do this we

will add an extra non-terminal to our grammar which will represent symbolic values. Our grammar will then look like

$\langle sym \rangle ::= \alpha \mid \beta \mid \gamma \mid \ldots$

$\langle int \rangle ::= 0 \mid 1 \mid -1 \mid \ldots$

$\vdots$

$\langle aexp \rangle ::= \langle sym \rangle \mid \langle int \rangle \mid \langle id \rangle \mid \ldots$

$\vdots$

Furthermore, we must extend the capabilities of our environment, so that it does not only map to Integer values, but instead to arbitrary expressions over integers and symbolic values. Furthermore we must extend the return type of functions as well. These too, will be allowed to return arbitrary expressions over integers and symbolic values.

Representing the path constraint

Since the path constraint is simply a conjunction of expression on the form $(\langle aexp \rangle op \langle aexp \rangle)$ with $op \in \{>, ==\}$, we will simply represent the PC as a list of such expressions. So if $PC = true \wedge e1 \wedge e1 \wedge \ldots \wedge e_n$, we will represent it as the list $L_{pc} = [true, e1, e2, \ldots, e_n]$

Chapter 4

Further extensions

Chapter 5

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] 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`.