Symbolic Execution(Working title)

Aarhus Universitet

Søren Baadsgaard

March 27, 2019

Abstract

Contents

1 Introduction **2**

2 Motivation **3**

3 Principles of symbolic execution **5**

3.1 Symbolic executing of a program 5

3.2 Execution paths and path constraints 6

3.3 Constraint solving 8

3.4 Limitations and challenges of symbolic execution 9

3.4.1 The number of possible execution paths 9

3.4.2 deciding satisfiability of *path-constraints* and constraint solving 10

4 Principles of Concolic execution **11**

5 Defining the SIMPL language **12**

6 Symbolic execution of SIMPL **13**

6.1 description 13

6.2 Introducing the ***SImPL*** language 13

6.2.1 Interpreting ***SImPL*** 15

6.2.2 Symbolic interpreter for ***SImPL*** 16

7 Concolic execution of SIMPL **18**

8 Conclusion **19**

A Source code **20**

B Figures **21**

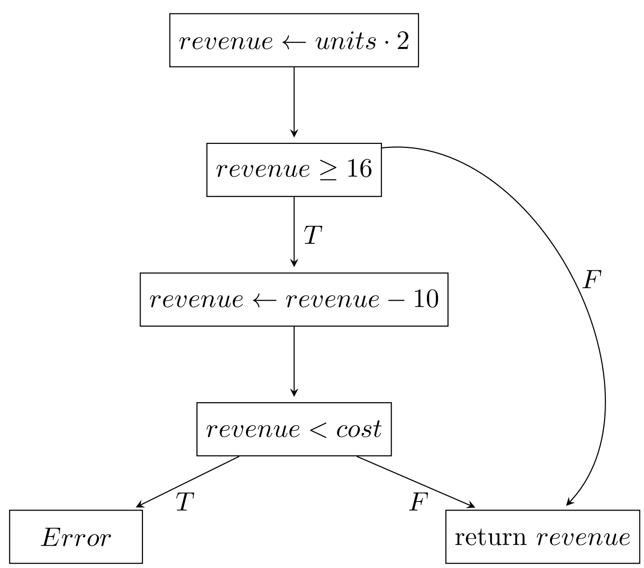
Chapter 1

Introduction

Chapter 2

Motivation

Consider the following program that takes integer inputs *units* and *cost*



We would like to know if this program ever fails, so we have to figure out if there exist integer inputs for which the program reaches the *Error* statement. We might try to run the program on different input values, e.g $(units = 8, cost = 5)$, $(units = 7, cost = 10)$. Running the program with these inputs, does not crash the program, but we are still not convinced that it wont crash for some other input values. By observing the program long enough, we realize that the input must satisfy the following two constraints to crash:

$$
\begin{aligned}
units \cdot 2 &\geq 16 \\
units \cdot 2 &< cost
\end{aligned}
$$

which is the case 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.

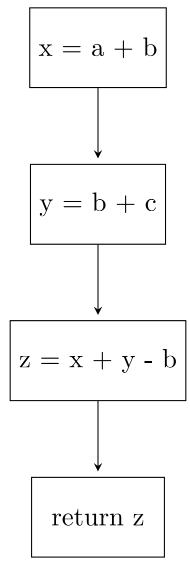
Chapter 3

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 one. 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:



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 $[q_1, q_2, \ldots, q_n]$ 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. We only execute along a path if $q_1 \wedge \ldots \wedge q_n$ is *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*.

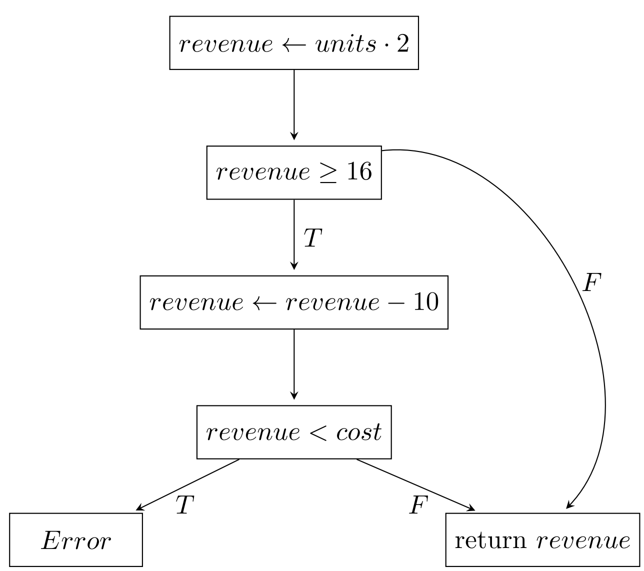
Whenever we reach a conditional statement with condition q, we consider the two following expressions:

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

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**: We add q to the *path-constraint*, and we continue the execution along path corresponding to the condition evaluating to $true$.
- **Only the second expression is satisfiable**: We add $\neg q$ to the *path-constraint*, and we continue along the path corresponding to the condition evaluating to $false$.
- **Both expressions are satisfiable**: In this case, the execution can follow 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 as the final element and the other will have $\neg q$. These two executions will now follow two 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*:



If we assign symbolic values α and β to *units* and *cost* respectively, we 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 add q_1 to the current *path-constraint* and continue the execution along the T path, and start a new execution that follows the F path. This execution will have the same variable bindings, but the *path-constraint* will receive $\neg q_1$ instead. This execution reaches a terminal statement, in which we return $\alpha \cdot 2$. The first execution assigns $2 \cdot \alpha - 10$ to *revenue* and then reach another conditional statement with condition $2 \cdot \alpha - 10 < \beta$. We consider the following expressions:

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

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

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 *Error* statement, 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 that contains branching by representing different execution paths by *path-constraints*, which is a list of boolean expressions that the input values must satisfy in order to follow a given path. Since we only consider *path-constraints* that is satisfiable, we know that for each path we discover, there exists concrete input values that will follow this exact path during a concrete execution. This means that we can solve the system of constraints that is contained in the *path-constraint* and obtain a single test case that represents all possible input values that satisfy the same system of constraints.

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

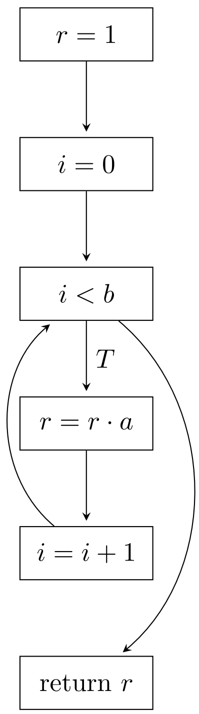
By solving for α and β, we obtain the set of test cases $\{(7, \beta), (8, 6), (8, 7)\}$, that covers all possible execution paths. Note that we have excluded a concrete value for β in the first test case, because the path represented by this *path-constraint* does not depend on the value of β.

3.4 Limitations and challenges of symbolic execution

So far, we have considered symbolic execution of very nicely behaving programs, in terms of the number of possible execution paths and the types of constraints that must be solved. In this section we will cover the challenges that arise when we consider programs that does not behave so nicely.

3.4.1 The number of possible execution paths

Since each conditional statement in a given program can result in up to 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 can illustrate this by considering the following program that implements the power-function for integers a and b, with symbolic values α and β for a and b:



This program contains a $While$-statement with condition $q = i < b$. The $k'th$ time we reach this statement we will consider the following two expressions:

1. $true \wedge (0 < \beta) \wedge (1 < \beta) \wedge \ldots \wedge (k - 1 < \beta)$
2. $true \wedge (0 < \beta) \wedge (1 < \beta) \wedge \ldots \wedge \neg(k - 1 < \beta)$.

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*** and constraint solving

Chapter 4

Principles of Concolic execution

Chapter 5

Defining the SIMPL language

Chapter 6

Symbolic execution of SIMPL

6.1 description

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

6.2 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

SImPL supports two different types of expressions, arithmetic expressions and boolean 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, and therefore functions must return integer values. **boolean expressions** consists of the boolean values $true$ and $false$, as well as comparisons of arithmetic expressions.

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.

6.2.1 Interpreting ***SImPL***

In order to work with ***SImPL*** , we have build a simple interpreter using the *Scala* programming language. To keep track of our program state, we define a map

$$env : \langle Id \rangle \to \mathbb{N} \tag{6.1}$$

that maps variable names to integer values. When interpreting a statement or an expression, this map will be passed along. Whenever we interpret a function call, we make a copy of the current environment to which we add the call-parameters. This copy is then passed to the interpretation of the function body, in order to ensure that functions are side-effect free.

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

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

from function names to top-level function declarations, as well as a root statement. To interpret the program we simply traverse the AST starting with the root statement.

return values

In order to handle return types, we extend the implementation of the grammar with a non-terminal

$\langle value \rangle$ ::= IntValue | BoolValue | Unit.

Arithmetic expressions will always return an $IntValue$ and Boolean expressions will always return a $BoolValue$. $Unit$ is a special return value which is reserved for *while*-statements.

6.2.2 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

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

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

$\vdots$

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

$\vdots$

Determining feasible paths

In order to determine which execution paths are feasible, we use the **Java** implementation of the *SMT-solver* **Z3**.

Return values

We extend our return values with the terminal $SymValue$ which contains expressions of the type *Expr* from **Z3**, over integers and symbolic values.

Representing the path constraint

To represent the *path-constraint* we implement a class $PathConstraint$ which contains a boolean formula $f = BoolExpr \wedge BoolExpr \wedge ...$ where each expression of type *BoolExpr* is a condition that the input values must satisfy.

Representing the program state

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. Therefore we define environment as a map

$$m : \langle Id \rangle \rightarrow SymValue$$

from variable identifiers to values of type $SymValue$.

Execution strategy

The first execution strategy that we implement is a naive approach, where all feasible paths will be explored in *Depth-first* order, starting with the *else*-branch. Note that our definition of *feasible* is any path that we can determine to be satisfiable. This means that *path-constraints* that **Z3** cannot determine the satisfiability of, will be regarded as infeasible, and ignored. This strategy is sufficient for small *finite* programs, but scales badly to programs with large recursion trees, and it runs forever on programs with infinite recursion trees.

Chapter 7

Concolic execution of SIMPL

Chapter 8

Conclusion

Appendix A

Source code

Appendix B

Figures

Bibliography

[1] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56:82–90, 02 2013. doi: 10.1145/2408776.2408795.

[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`.