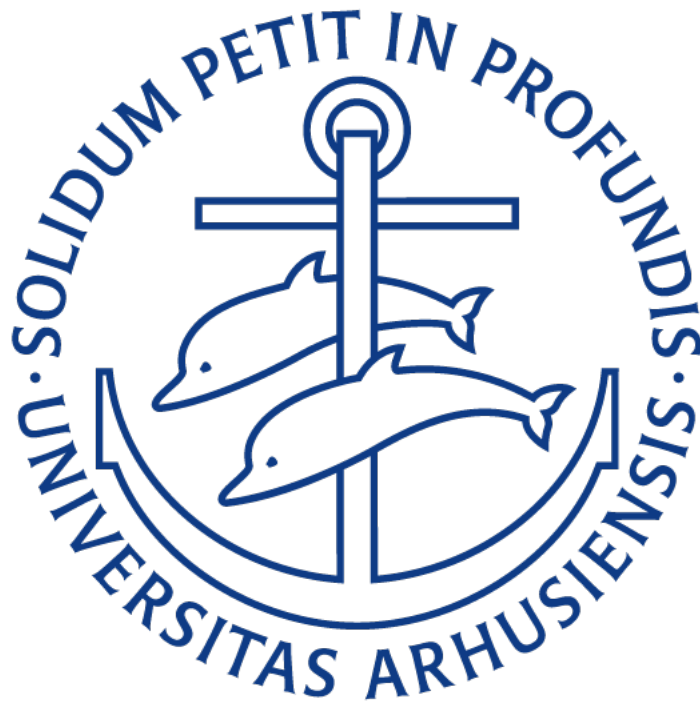


# Symbolic Execution(Working title)

Aarhus Universitet



Søren Baadsgaard

February 5, 2019

## Abstract

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Summary of theory</b>	<b>3</b>
<b>3</b>	<b>Basic symbolic execution for the <i>SImPL</i> language</b>	<b>4</b>
3.1	description . . . . .	4
3.2	Introducing the <i>SImPL</i> language . . . . .	4
3.2.1	Interpreting <i>SImPL</i> . . . . .	5
<b>4</b>	<b>Further extensions</b>	<b>6</b>
<b>5</b>	<b>Conclusion</b>	<b>7</b>
<b>A</b>	<b>Source code</b>	<b>8</b>
<b>B</b>	<b>Figures</b>	<b>9</b>

# Chapter 1

## Introduction

## Chapter 2

### Summary of theory

## 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* (**S**imple **I**mpерative **P**rogramming **L**anguage) 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