Utrecht University

MASTER THESIS

Evaluating the Effectiveness of an OOX based Symbolic Execution Engine

Author:
Tjeerd SMID
5984726

Supervisors:
Dr. S.W.B. PRASETYA
Prof. dr. G. KELLER

A thesis submitted in fulfillment of the requirements
for the degree of Master of Science

in the

Department of Information and Computing Sciences

July 31, 2023

UTRECHT UNIVERSITY

Abstract

Graduate School of Natural Sciences

Department of Information and Computing Sciences

Master of Science

Evaluating the Effectiveness of an OOX based Symbolic Execution Engine

by Tjeerd SMID

Symbolic execution is a program testing technique from the software verification domain. It involves symbolically modeling and testing all possible execution paths of a program against a set of constraints. The three main challenges of symbolic execution are (1) **Memory modeling**; (2) **Execution path explosion**; and (3) **Constraint solving**.

In this thesis, we present a Symbolic Execution Engine. This engine operates on the OOX language and is equipped with eleven different heuristics. These heuristics aim to improve efficiency in handling the second and third challenges of symbolic execution.

We conducted an extensive experiment with our Symbolic Execution Engine. We used the benchmarking tools and a set of 81 benchmarking programs from the Software Verification Competition, and seven comparable verification tools, with the aim of investigating the effectiveness of the heuristics and substantiating the claims of effectiveness, soundness, and completeness of our Symbolic Execution Engine.

Contents

1 Introduction **1**

2 Background **2**

2.1 Testing 2

2.1.1 Unit testing 2

2.1.2 Property-Based Testing 3

2.1.3 Symbolic Testing 4

2.2 Related Work 5

2.2.1 Constraint Solving 6

2.2.2 Path Explosion 8

2.2.3 Software Verification Tools 9

2.3 Intermediate Verification Language OOX 10

3 The Symbolic Execution Engine **12**

3.1 Parsing 13

3.2 CFG Generation 13

3.2.1 Statements to Control Flow Graph (CFG) 14

3.2.2 Program to CFG 17

3.3 Symbolic Execution 19

3.3.1 Control Flow 19

3.3.2 Expression 19

3.3.3 Path Constraints 20

3.3.4 Satisfiability Solving 21

3.3.5 Symbolic Reference 22

3.3.6 Reference Values 22

3.3.7 Stack 25

3.3.8 Heap 25

3.3.9 Algorithm 28

3.3.10 Updating Symbolic State 30

3.4 Heuristics 34

3.4.1 Pruning 34

3.4.2 Interval Inference 35

3.4.3 Extended Expression Evaluation 41
3.4.4 Expression Caching 43
3.4.5 Parallel Solving 44

4 Results **45**

4.1 Benchmark Setup 45
4.2 Heuristics 48
4.2.1 Expression Evaluation 50
4.2.2 Pruning 54
4.2.3 Expression Caching 55
4.2.4 SMT Solver 57
4.3 Verifier Comparison 59
4.4 Soundness and Completeness 62
4.4.1 Soundness 62
4.4.2 Completeness 64

5 Conclusion **65**

6 Future Work **67**

List of Acronyms

SEE Symbolic Execution Engine

BFS Breath-First Search

SMT Satisfiability Modulo Theories

SMT-COMP International SMT Competition

SV-COMP Competition on Software Verification

IVL Intermediate Verification Language

CFG Control Flow Graph

AST Abstract Syntax Tree

SS Symbolic State

EEE Extended Expression Evaluator

II Interval Inferer

II2 Interval Inferrer (two iterations)

PP Probabilistic Pruner

AP Adaptive Pruner

CP Constant Pruner

EC Expression Cacher

NEC Normalized Expression Cacher

Chapter 1

Introduction

From the microprocessor managing our ovens to the supercomputers occupying multiple football fields, digital computers are ubiquitous, and so is the software running them. A software failure in your oven's microprocessor may hinder the preparation of your lasagna tonight, but there are far worse problems caused by these failures. Failure of software for spacecraft, airplanes, medical devices or flash trading can and does result in millions of dollars in damages or even worse, loss of human life.

Knowing the damages that software failures can cause and the fallibility of the humans developing this software, the need for preventing software failures is clear. This need lies at the heart of many research fields such as requirement engineering and software testing & verification.

In this thesis, we consider one method, from the software testing & verification domain, that can be used to discover a subset of software failures: symbolic execution. We develop a Symbolic Execution Engine (SEE), named **Jip**[1], operating on the OOX programming language with a set of eleven heuristics. The main question we aim to address is:

Is a SEE operating on OOX a viable alternative to current program verification tools?

[1]The public repository containing the SEE can be found at https://github.com/tjausm/Jip

Chapter 2

Background

2.1 Testing

In this Chapter, we treat software testing as a search through all possible executions of a program. During this search, we look for one or more faulty executions, without knowing if there even is a faulty execution. In the following sections, we take a closer look at various ways in which we can do this exploration, to get an understanding of what testing is, how we can define what a faulty execution is and to motivate the choice of testing software with symbolic execution. We use the program seen in listing 1 as a running example throughout Chapter 2.

2.1.1 Unit testing

One of the more straightforward methods of testing would be constructing a set of inputs and corresponding outputs. For example, consider the program `mod(a,b)` shown in figure 1. To demonstrate the correctness of this program with unit testing we could assume there are two cases of inputs: `a < b` and `a >= b`. And if our program can solve one instance of this case, this approach assumes it can solve all other instances of this case. Possible test inputs covering the two previously mentioned cases are `(12, 88)` and

[1] We assume `a >= 0 && b > 0` for the sake of simplicity

```
1  static int mod(int a, int b){
2      assume a >= 0;
3      assume b > 0;
4
5      res := a;
6
7      while (res >= b) {
8          res := res - b;
9      }
10
11     assert res == a % b;
12     return res;
13 }
```

LISTING 1: A program that writes the result of `a % b`[1] to `res`

`(73, 10)`, giving us the following two tests: `mod(12, 88) == 12` and `mod(73, 10) == 3`. With this trivial program, this method already poses problems, as we will see in section 2.1.3. But constructing inputs for all cases of less trivial programs is even more complex.

Unit testing also suffers from a bias. Programmers create a set of test inputs for each case a program should cover. If a programmer knows of the case he writes a set of test inputs for, the programmer has probably covered that case in his program. If the programmer does not know of a case of inputs that the code should cover, the programmer has most likely not written a test input covering that case.

2.1.2 Property-Based Testing

A more sophisticated method that removes some of this aforementioned bias and does not force the developer to construct all cases is property-based testing. A software developer writes a predicate to which the output has to adhere (with respect to an input), and a testing library generates random data to test whether the predicate holds. For listing 1 we could have listing 2 as a predicate.

This method can be quite thorough, assuming that the predicate is well written and the random input values and sizes are non-trivial. But even in our very simple example the amount of possible inputs are 2^{32^2}. One iteration of random tests only confirms our program to be valid for the minuscule fraction of inputs that we have generated, leaving an enormous amount of inputs that could still break our predicate.

```
1 static bool isMod(int a, int b, int res){
2     return a < 0 || b <= 0 || a % b == res
3 }
```

LISTING 2: Predicate of a successful modulo application

2.1.3 Symbolic Testing

We argue that the problems described in the previous section arise from the instantiating variables when testing the example. Given that an integer can usually take on 2^{32} values, one can see how instantiating many of these, which happens in most meaningful programs, can lead to virtually infinite test cases. This makes it impossible to *completely test the program. A solution to this problem is avoiding the instantiation of variables, and instead, verifying whether an execution path of the program could break our predicate.*

To get an intuition of how we avoid instantiating variables we, again, take a look at listing 1. The CFG representing listing 1 can be seen in Figure 2.1. All possible executions of the `mod` *can be represented as a path through this CFG. To see whether it is correct without instantiating variables we would go over the paths we can take through the CFG in 2.1. We can intuitively see two 'types' of paths. Either* `res >= b` *and we traverse the loop* `n` *times or* `!res >= b` *and we never enter the while loop.*

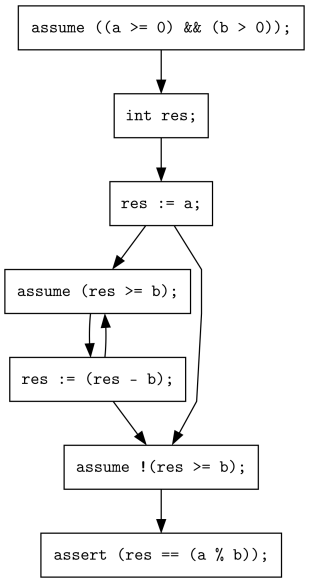

FIGURE 2.1: CFG[2]of listing 1

[2]CFG is generated by our tool Jip

Over the first path, where `res >= b`*, we keep subtracting* `b` *from* `res` *until* `!res >= b`*. Òver the second path, where* `!res >= b`*, we return* `res`*. Intuitively we see how both these paths should return the correct answer. This is because* `res` *is initialized to* `a` *and* `a % b` *should either return* `a` *if* `a < b` *or* `a - n * b` *such that* `0 <= a && a < b`*. Symbolic execution formalizes this process. In Chapter 3 we will extend on how our SEE goes about formalizing this process.*

2.1.3.1 Challenges of Symbolic Testing

In the last section, we proposed that not instantiating variables dramatically reduces the work needed to completely test a program. But symbolic testing comes with its own challenges[5]:

Memory *modeling values on the stack is manageable. But complexity grows as we start modeling references and their corresponding values on the heap. Does the nth element of an array exist? Do we even know that the arrays reference is not null? and so on.*

Constraint solving *, in this context, is an NP-hard problem. And although state-of-the-art solvers such as Microsoft's Z3 Satisfiability Modulo Theories (SMT) solver are very efficient, solving the constraints remain a very large part of the method's complexity.*

Path explosion *or state space explosion. Symbolic execution reasons about all execution paths of a program. The number of paths grows exponentially with each control structure our program contains e.g. adding an if-then-else branch doubles the amounts of paths, and most useful programming language contains even worse constructs (from the perspective path quantity), namely infinite ones e.g. while loops, arrays, for loops, recursion, etc.*

2.2 Related Work

Circa 1975 symbolic execution was simultaneously invented by several researchers [14][8]. Almost 50 years later, as of this writing, there are circa 1600 articles containing the phrase "symbolic execution" on Google Scholar. A simple Google search returns many tools to perform symbolic execution with languages such as C, Ruby, Javascript, and Java. Apart from the method's presence

in academia, Microsoft has reported that its symbolic testing tools play a big role in testing many of their application, r eporting that nearly 30% of all bugs found in Windows 7, were found using said technique [11]. Symbolic execution may not be as ubiquitous as unit testing. But the method has found its way into the mainstream, especially for testing critical applications.

In this thesis, the focus will lie on optimizing constraint solving and reducing the path explosion problem. Therefore, in the following subsections, we will expand on different methods from the literature to do this, discussed in subsections 2.2.1 and 2.2.2, and we will provide some background on various similar tools that already exist in subsection 2.2.3.

2.2.1 Constraint Solving

During symbolic execution, we use sets of constraints collected over execution paths to verify paths as either infeasible, valid, or containing an error. The most popular method to solve sets of constraints is via SMT solvers. An SMT solver can be called with a set of constraints as input, and the solver will determine if there exists an assignment of values to the variables in the constraints such that all the constraints are satisfied. Many SMT calls have to be made for each execution path, and deciding the satisfiability of a set of constraints is an NP-hard problem. As a result, constraint solving is one of the main factors that impact performance.

In the meta-analysis of Symbolic Execution conducted by Baldoni et al.[5], three categories of methods are presented to improve performance in constraint solving during symbolic execution. Each of these categories will be discussed, with particular emphasis on Constraint Reusing and Constraint Reduction, as several heuristics in this thesis belong to these categories.

Constraint reusing *aims at caching the results of previously solved constraints. When symbolic execution encounters a constraint, it first consults the cache to see if the constraint or a similar one has already been solved. If so, the cached solution can be reused.*

The methods of reusing can vary, from simple direct caching, where previously seen constraints are mapped to results, to more complex methods such as subsuming cache or incremental solving. The subsuming cache method aims to map subsuming constraints to each other in the cache. Meanwhile, incremental solving aims to persist the SMT solver state across calls, which allows the solver to reuse its internal state.

Constraint simplification *attempts to simplify the constraints to the extent that either the SMT solver is not required, or the task of solving the constraints is made easier for the SMT solver. Simplification can be done in a variety of ways such as evaluating concrete values, using domain-specific knowledge gained during the symbolic execution, or removing irrelevant constraints. These methods often also lead to better performance in constraint reusing.*

Augmenting constraint solving *to handle constraints that are difficult for conventional solvers. This category of optimization is aimed at extending the applicability and effectiveness of SMT solvers in scenarios where traditional constraint solving techniques may struggle.*

2.2.1.1 SMT Solvers

Because constraint solving greatly contributes to the complexity of symbolic execution, the performance of an SMT solver significantly influences the performance of SEEs. In this section, we will provide a brief overview of the landscape of SMT solvers and highlight the three solvers used in this thesis.

The most extensive comparative analysis of SMT solvers is during the annual International SMT Competition (SMT-COMP)[4]. In 2023, this competition hosted six different tracks ranging from testing solver performance on distributed systems to testing the performance of solvers reducing problems to their minimal unsat core. However, the most important category, in the context of this thesis, is the single query track, comparing the performance of solvers on solving a single satisifiability problem, since more extensive usage of solvers with parallel or distributed verification is beyond the scope of this thesis. The single query track is divided into 95 different categories, of which we have summarized the best-performing verifiers in table 2.1.

Solver	cvc5	Bitwuzla	Yices2	smtinterpol	Z3	YicesQS	OpenSMT	STP	Vampire
Categories Won	49	18	13	4	3	2	3	2	1

TABLE 2.1: Performance of SMT solvers in the Single Query track in 2023

The three solvers used in this thesis are CVC4, Yices2, and Z3:

CVC4[1] *is a solver developed as a joint project by Stanford University and The University of Iowa. While this solver has now been succeeded by cvc5, it was the best-performing solver in the Single Query track during the last SMT-COMP in which it participated, in 2019.*

Yices2[2] *is a solver developed by the Stanford Research Institute. It has consistently been one of the top-performing solvers in the SMT-COMP for several years.*

Z3[3] *is a solver developed by Microsoft Research. Though it does win in some categories of the SMT-COMP, it performed less well than Yices2 and cvc4 in the Single Query track in 2019. Z3 is the most popular SMT solver*[3]*. Consequently, Z3 has the most extensive support in terms of documentation, library, and API availability.*

2.2.2 Path Explosion

The path explosion problem refers to the exponential growth in the number of execution paths that symbolic execution needs to explore as the size and complexity of the program increase, which poses a problem since many more paths have to be considered during symbolic execution when searching for errors.

In the meta-analysis of Symbolic Execution conducted by Baldoni et al.[5], six categories of popular methods for combating the path explosion problem are presented. Each category will be discussed, with particular emphasis on path feasibility, as several heuristics in this thesis belong to this category.

Path feasibility. *In symbolic execution, a distinction is made between feasible and infeasible execution paths. Feasible execution paths are those paths for which a set of input values exists that will result in an execution over the path. On the other hand, infeasible execution paths are paths for which no set of input values exists that will result in an execution over*

[3] A Google Scholar search for 'z3 SMT' results in 17700 results, whereas 'cvc4 SMT' and 'yices2 SMT' resulted in 2400 and 430 results respectively.

the path. For instance, a path containing the condition `if (x > 5 && x < 3)` *will never be executed, since there is no value for* `x` *for which both* `x > 5` *and* `x < 3`.

Verifying an infeasible path during symbolic execution is redundant since a bug cannot occur in an execution path that cannot exist. Contrastingly, eagerly verifying all constraints to prevent the exploration of infeasible paths can be more expensive than exploring the infeasible paths themselves. In the literature, the exploration of paths without checking for infeasibility is called lazy constraint evaluation. On the other hand, the approach of checking all constraints is referred to as eager constraint evaluation[5].

One of the more popular methods to reduce the cost of recognizing infeasible paths was proposed by Calcagno et al.[10]. They proposed reducing the constraints to a minimal unsat core, the smallest set of constraints from an execution path that is proved to be unsatisfiable. By memoizing this unsat core, other infeasible paths can be recognized without invoking an SMTsolver.

Loop and function Summarization *attempts to reduce the path explosion problem by abstracting and representing the effect of loops or functions, as summaries. These summaries capture the essential behavior of the loop or function and can be reused whenever the loop or function is encountered again.*

Path subsumption *tries to discard newly explored paths if the behavior of a path is already covered by a previously explored path, known as a subsuming path.*

State merging *attempts to combine states, storing their constraints as disjunctions, resulting in a single, more complex state that represents multiple paths.*

Under-constrained Symbolic Execution *tries to analyze a smaller segment of a program in isolation. This allows symbolic execution to conduct a more complete analysis of the segment without considering all other paths throughout the entire program.*

2.2.3 Software Verification Tools

Similarly to SMT-COMP, the Competition on Software Verification (SV-COMP) is the largest annual competition for software verification tools. In the 2023 edition, 57 verification tools participated, wherein eight verifiers competed in the Java competition, while the remaining 49 participated

in the C++ competition. Due to the scope of this thesis, only the Java verifiers will be used in the comparative analysis.

The eight verifiers use four different methods or combinations of methods to verify a program. In Table 2.2, we present the method each of the eight Java verifiers uses, as per their respective publications[25][18] [23] [9][13] [17] [20]. Note that the SEE developed for this thesis solely utilizes symbolic execution.

Verifier	Concrete execution	Symbolic execution	Model checking	Bounded model checking
Coastal	X	X		
Gdart	X	X	X	
Java Ranger		X	X	
Jayhorn		X	X	
JBMC		X		X
Jdart	X	X		
MLB				X
SPF		X	X	

TABLE 2.2: Verification methods used by all Java verifiers competing in SV-COMP

2.3 Intermediate Verification Language OOX

The SEE that will be presented in Chapter 3 operates on OOX. Which is an Intermediate Verification Language (IVL) designed by Stefan Koppier[15]. IVLs are programming languages intended to be an intermediate between higher-level languages and the verification tool. Using an IVL allows the SEE to operate on a simpler model while still being able to verify programs written in any language, as long as they can be parsed to OOX.

OOX has a strong type system with concurrency, classes, and objects as its first-class citizens. Its object orientation makes it well-suited to model languages such as C# and Java. The formal semantics of the language are also available[15].

Implemented OOX features *The SEE described in this Chapter operates on OOX without its concurrency features, as described in Table 2.3.*

ty v	Introduce a variable *v* of type ry with a default value
lhs := *rhs*	Assign the evaluated value of *rhs* to *lhs*
$o.f(e1, ..., e_n)$	Invoke method f of an object o
`assert` e	Assert that the expression e holds.
`assume` e	Assume that the expression e holds.
`while`(e) S	Execute S while the guard e evaluates to true.
`if`(e) S_1 `else` S_2	Execute S while the guard e evaluates to true.
`continue`	Jump from within a loop to its guard
`break`	Break the execution of a loop.
`return` e	Return to the caller, passing the value of e
$S_1 ; S_2$	Chain consecutive statements

TABLE 2.3: Set of *Statements* supported by Jip (taken from Koppier et al. [22])

Chapter 3

The Symbolic Execution Engine

The SEE operates in three phases as outlined in Figure 3.1. The first two phases return either an error or a transformed OOX program, the last phase can return either an error or the verification result. As no static analysis after parsing, semantic errors can arise during both the CFG generation and symbolic execution phases. For instance, semantic errors that prevent the SEE from generating a CFG, such as calling a non-existent method, will result in a semantic error during CFG generation. Likewise, errors that prevent the SEE from executing, such as evaluating an ill-typed expression (e.g. `false + 1`) will result in a semantic error during the symbolic execution.

The parsing process will be elaborated upon in section 3.1. The CFG generation process will be explained in section 3.2. Lastly, the symbolic execution will be discussed in section 3.3.

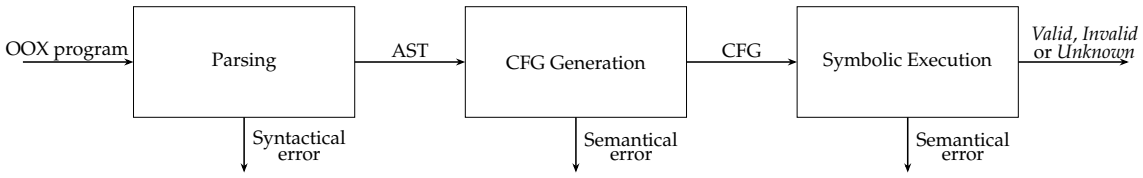

FIGURE 3.1: Overview of SEE architecture

3.1 Parsing

The first phase of symbolically executing an OOX program consists of parsing an OOX program. Parsing an OOX program returns an Abstract Syntax Tree (AST). No additional transformations are applied during this phase.

3.2 CFG Generation

The second phase consists of generating a CFG from program $p \in \mathit{Program}$, generated during the parsing phase, as outlined in 3.1.

$$\begin{aligned} \mathit{Program} &::= \mathit{Class}^{+} \\ \mathit{Class} &::= (\mathit{Identifier},\ \mathit{Method}^{*}) \\ \mathit{Method} &::= (\mathit{Paremeter}^{*},\ \mathit{Statements}) \\ \mathit{Statements} &::= \mathit{Statement} \mid \mathit{Statement}\ \mathit{Statements} \\ \mathit{Identifier} &::= \mathit{String} \end{aligned} \tag{3.1}$$

The CFG is a directed graph (N, E) representing the control flow paths of a program. An edge from node n_1 to node n_2 is denoted as $n_1 \rightarrow n_2$. A node $n \in N$, as shown in 3.2, consists of a unique label and one of the five actions: (1) entering the `Main.main` function[1], the starting point of the CFG; (2) leaving the `Main.main` function, the end point of the CFG; (3) entering the subgraph of methods, static methods, and constructors (which we categorize as procedures); (4) leaving the subgraph of a procedure; and (5) executing a statement.

[1] The SEE always assumes `Main.main` to be the entry method

$$
\begin{aligned}
N &= \mathit{Label} \times \mathit{Action} \\
\mathit{Action} &= \mathit{EnterMain} \\
&\mid \mathit{LeaveMain} \\
&\mid \mathit{EnterProcedure} \\
&\mid \mathit{LeaveProcedure} \\
&\mid \mathit{Execute}(\mathit{Statement})
\end{aligned}
\tag{3.2}
$$

3.2.1 Statements to CFG

The CFG generation is a function $s2c$ that recursively evaluates statements returning a triple: a CFG c, the set of start node(s) of c, and the set of end node(s) of c.

$$s2c : \mathit{Statements} \rightarrow (\mathit{CFG} \times \{N\} \times \{N\})$$

Function *s2c* also maintains two environments:

- **Type Environment.** This environment maps object names to their respective class names. If the passed identifier is not in the environment, the function serves as the identity function, to account for static method calls.

  $$\mathit{Type}_{env} : \mathit{Identifier} \rightarrow \mathit{Identifier}$$

- **Procedure Environment.** This environment maps the identifiers of methods, static methods, and constructors (which we categorize as procedures) to the subgraphs and start and end nodes of these subgraphs. The first time the subgraph of a procedure is retrieved, it is recursively generated with the function *s2c*, a node of the form $(l, \mathit{EnterProcedure})$ is prepended, and a node of the form $(l, \mathit{LeaveProcedure})$ is

appended, acting as the start and end nodes of the generated subgraph. All subsequent retrievals return the previously generated start and end nodes.

$$proc_{env} : (Identifier, Identifier) \rightarrow (CFG \times N \times N)$$

We define function $fl()$ that generates a unique label for each node and a set of auxiliary functions to combine CFGs, nodes and edges.

$$(N_1, E_1) + (N_2, E_2) = (N_1 \cup N_2,\ E_1 \cup E_2)$$
$$(N_1, E_1) + N_2 = (N_1 \cup N_2,\ E_1)$$
$$(N_1, E_1) + E_2 = (N_1,\ E_1 \cup E_2)$$

In the following six paragraphs we will elaborate on how function *s2c* recursively generates a CFG. Each paragraph will discuss one case of *s2c*, starting with the formal definition of the function for that case, followed by a piece of example code and its corresponding CFG.

if-then-else. Let $s2c(\texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2) = res$. We define nodes $n_1 = (fl(),\ Execute(\texttt{assume}\ e))$ and $n_2 = (fl(),\ Execute(\texttt{assume}\ \neg e))$ to represent the branch conditions and recursively evaluate the bodies of the if-then-else statement: $s2c(s_1) = (cfg_1,\ start_1,\ end_1)$ and $s2c(s_2) = (cfg_2,\ start_2,\ end_2)$. This allows us to define *res* as $(cfg_1 + cfg_2 + \{n_1, n_2\} + \{n_1 \rightarrow n_s : n_s \in start_1\} + \{n_2 \rightarrow n_s : n_s \in start_2\},\ \{n_1, n_2\},\ end_1 \cup end_2)$.

```
1 if (x) {
2     y := 1;
3 }
4 else {
5     y := 2;
6 }
```

→

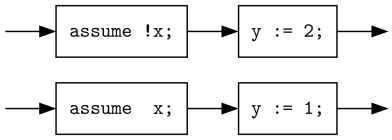

FIGURE 3.2: if-then-else to CFG

While. Let $s2c(\texttt{while}(e)\ s) = res$. We define nodes $n_1 = (fl(),\ Execute(\texttt{assume}\ e))$ and $n_2 = (fl(),\ Execute(\texttt{assume}\ \neg e))$ to represent the branch conditions and recursively evaluate the body of the while statement as $s2c(s) = (start, end, cfg)$. This allows us to define res as $(cfg + \{n_1, n_2\} + \{n_1 \rightarrow n_s : n_s \in start\} + \{m \rightarrow n : m \in end,\ n \in \{n_1, n_2\}\},\ \{n_1, n_2\},\ \{n_2\})$.

```
1  while (x) {
2      y := y + 1;
3  }
```

→

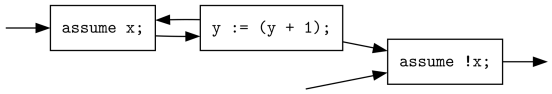

FIGURE 3.3: While to CFG

Procedure Invocation. Let $s2c(id.f(args)) = proc_{env}(type_{env}(id), f)$.

```
1  static void main(){
2      Main.f();
3  }
4  static void f(){
5      ;
6  }
```

→

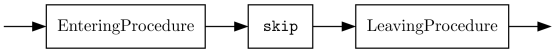

FIGURE 3.4: Procedure Invocation to CFG

Procedure Call. Let $s2c(id := id.f(args)) = res$. We define node $n = (fl(), Execute(id := \texttt{retval}))$ and evaluate $proc_{env}(type_{env}(id), f) = (cfg, start, end)$ to retrieve the CFG, start and, end node(s) of the procedure. This allows us to define res as $(cfg + \{n\} + \{end \rightarrow n\}), \{start\}, \{n\})$.

```
1 static void main(){
2    h := Main.two();
3 }
4 static int two(){
5    return 2;
6 }
```

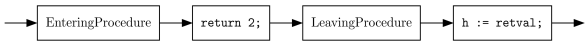

FIGURE 3.5: Procedure Call to CFG

Recursive Statement. Let $s2c(s_1\ s_2) = res$, let $s2c(s_1) = (start_1,\ end_1,\ cfg_1)$ and let $s2c(s_2) = (start_2,\ end_2,\ cfg_2)$. This allows us to define res as $s2c(\ cfg_1 + cfg_2 + \{e \rightarrow s : e \in end_1,\ s \in start_2\},\ start_1,\ end_2)$.

```
1 s1; s2
```

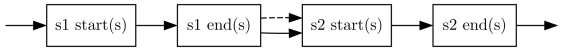

FIGURE 3.6: Recursive Statement to CFG

Otherwise. Let $s2c(s) = res$. Then we define node $n = (fl(),\ Execute(s))$ allowing us to define res as $((n, \emptyset), \{n\}, \{n\})$.

```
1 y := y + 1
```

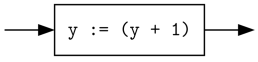

FIGURE 3.7: Otherwise to CFG

3.2.2 Program to CFG

To generate a CFG c from p we use the following procedure: (1) defining $n_s = (fl(),\ EnterMain)$ and $n_e = (fl(), LeaveMain)$, the start- and end node; (2) retrieving the `Main.main` method m from p; (3) using body b of m to generate a CFG $s2c(b) = (cfg, n_{sb}, n_{eb})$; and (4) defining c by combining generated nodes and edges $c = cfg + \{n_s, n_e\} + \{(n_s \rightarrow n_{sb}), (n_{eb} \rightarrow n_e)\}$ and

```
1  class Main {
2     static void main
3     (int x, int y)
4     {
5        int z;
6        if (x >= y)
7           z := x;
8        else
9           z := y;
10    }
11 }
```

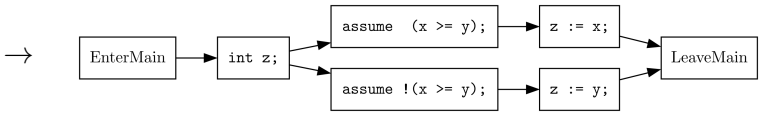

FIGURE 3.8: Program to CFG

3.3 Symbolic Execution

In this section, we start by explaining how the SEE symbolically represents all distinct components of a program execution, such as expressions, the stack, the heap, and so forth. Then, in section 3.3.9, we discuss the main algorithm that drives the symbolic execution. Section 3.3.10 will then combine all the previously explained components within the function *update_state*, which is the main function in the symbolic execution algorithm.

3.3.1 Control Flow

Verdict We define *Verdict* represening the three possible outcomes of symbolic execution (1) the program being error-free; (2) the program containing an error; or (3) the symbolic execution being unable to fully verify a program despite not encountering any errors.

$$\mathit{Verdict} = \mathit{Valid} \mid \mathit{Invalid} \mid \mathit{Unknown}$$

Infeasibility We define the unit type *Infeasible* to denote the Infeasibility of an execution path.

3.3.2 Expression

In the context of symbolic execution, an expression is the Expression type of OOX as outlined by Koppier [15] where all variables are substituted, resulting in an expression with only symbolic variables and symbolic references. Function *substitute* is used to evaluate an OOX expression to the expression used in the SEE. For instance, if the SEE encounters an expression `a == 4` with the following stack: $[a \mapsto \texttt{b} + 2]$, the SEE will evaluate $\mathit{substitute}([a \mapsto \texttt{b} + 2], \texttt{a} == 4)$ resulting in the expression `b + 2 == 4`. Note that `b` is a symbolic value in this expression.

The expression is defined as follows:

$$
\begin{array}{ll}
E \in \mathit{Expression} & ::= \mathit{lit}(z) \mid \mathit{lit}(b) \\
& \quad \mid \mathit{ref}(sr) \\
& \quad \mid \mathit{var}(id) \\
& \quad \mid \texttt{null} \\
& \quad \mid \mathit{unop}(E_1, \oplus) \\
& \quad \mid \mathit{binop}(E_1, \otimes, E_2) \\
& \quad \mid \mathit{sizeof}(i) \\
& \quad \mid \mathit{quantifier}(i_1, i_2, i_3, E, \odot) \\
z & ::= \mathbb{Z} \\
b & ::= \texttt{true} \mid \texttt{false} \\
sr & ::= \mathit{SymRef} \\
\\
id & ::= \mathit{String} \\
\\
\oplus \in \mathit{UnaryOperators} & ::= ! \quad \mid - \\
\otimes \in \mathit{BinaryOperators} & ::= * \quad \mid \backslash \quad \mid \% \quad \mid + \quad \mid - \\
& \quad \mid < \quad \mid <= \mid > \quad \mid >= \mid == \\
& \quad \mid != \mid \&\& \mid \mid\mid \mid ==> \\
\odot \in \mathit{quantifier} & ::= \texttt{forall} \mid \texttt{exists}
\end{array}
$$

3.3.3 Path Constraints

We define *PathConstraints*, representing the constraints collected over an execution path during the symbolic execution.

$$
\begin{array}{l}
\mathit{PathConstraints} = \mathit{Constraint}^* \\
\quad \mathit{Constraint} = \mathit{assume}(\mathit{Expression}) \mid \mathit{assert}(\mathit{Expression})
\end{array}
$$

Functions *conjunct* and *combine*, outlined in Formula 3.3, are defined for the path constraints. The interpretations of these functions are as follows: for any execution path with path constraints pc, if the expression $conjunct(pc)$ is satisfiable, the execution path is feasible. Moreover, if $\neg combine(pc)$ is satisfiable, then we have found an invalid execution path.

$$
\begin{aligned}
conjunct(pc) &= \begin{cases} True & \textbf{if } pc = [] \\ e \wedge conjunct([a_2, \ldots, a_n]) & \textbf{if } pc = [\texttt{assert}(e), a_2, \ldots, a_n] \\ e \wedge conjunct([a_2, \ldots, a_n]) & \textbf{if } pc = [\texttt{assume}(e), a_2, \ldots, a_n] \end{cases} \\
combine(pc) &= \begin{cases} True & \textbf{if } pc = [] \\ e \wedge conjunct([a_2, \ldots, a_n]) & \textbf{if } pc = [\texttt{assert}(e), a_2, \ldots, a_n] \\ e \implies conjunct([a_2, \ldots, a_n]) & \textbf{if } pc = [\texttt{assume}(e), a_2, \ldots, a_n] \end{cases}
\end{aligned}
\tag{3.3}
$$

3.3.4 Satisfiability Solving

The SEE interfaces with the SMT Solver using function *satisfiable*. This function returns either a mapping from identifiers to literals that satisfies the passed expression or $\perp$ if there exists no such mapping.

$$satisfiable : Expression \rightarrow (Identifier \rightarrow Literal) \mid \perp$$

Three SMT solvers are supported: (1) CVC4; (2) Yices2; and (3) Z3. The selection of these solvers was motivated by the availability of documentation and APIs for them in Rust, the language in which the SEE has been implemented.

3.3.5 Symbolic Reference

The symbolic reference is defined as the sum type *SymRef*, which is either Symbolic or Concrete depending on whether the reference is an argument to the `Main.main` method or is initialized during the symbolic execution. The label and the reference value in *SymRef* serve to encode the reference within the path constraints and to initialize its corresponding value. For instance, given a symbolic reference with label l and reference value rv, should the path constraints contain the expression l `!= null`, the value rv can be initialized on the heap. The implementation of this is discussed in further detail in Subsection 3.3.8.

$$SymRef : Symbolic(label, ref, refvalue) \mid Concrete(ref)$$

3.3.6 Reference Values

In OOX, a symbolic reference can point to two types of values: arrays and objects. In the following sections we will discuss how these are represented.

3.3.6.1 Array

Arrays are represented by the tuple (a, l), where a denotes a mapping from expressions to expressions, and l is an expression representing the length of the array. The mapping a initializes its values lazily, returning either concrete or symbolic values based on whether the array is an argument to the `Main.main` method or is initialized during the symbolic execution. Expression l, denoting the length of the array, is used throughout the symbolic execution to verify that an array access or update operation will never go out of bounds. This design allows arrays to have a symbolic length.

For instance, when an array `Int[] arr` is passed as an argument to the `Main.main` method of the symbolic execution and `arr[0]` has not been assigned a value, a fresh symbolic integer is returned if the array's length is one or greater. On the other hand, if `arr` was initialized during the symbolic execution and no value was assigned to `arr[0]`, the default

integer value of 0 is returned, provided the array's length is at least one. Likewise, upon accessing an array at index i with symbolic length l, the SEE will verify that under the path constraints expression `i < l` always holds.

We define three operations on arrays: (1) $init_{arr}$; (2) $update_{arr}$; and (3) $access_{arr}$. The first, $init_{arr}$, takes the type and length and initializes the array on the heap and returns its *SymRef* as an expression. The other two, $update_{arr}$ and $access_{arr}$, are outlined in Algorithms 1 and 2.

Algorithm 1: Update array

Input: Array tuple (a, l), path constraints pc, index expression i, and the expression to be inserted, v
Output: Updated array or a verdict

$pc_\wedge \leftarrow conjunct(pc)$;
$m \leftarrow satisfiable(pc_\wedge \wedge i = id_1)^a$;
if $m \neq \bot$ **then**
  if $satisfiable(pc_\wedge \wedge i = id_2 \wedge i \neq m(id_1)) = \bot$ **then**
    $i \leftarrow m(id_1)$;
  end
end
if $satisfiable(pc_\wedge \wedge i \geq l)^b$ **then**
  if $satisfiable(pc_\wedge)^c$ **then**
    return *Invalid*;
  else
    return *Infeasible*;
  end
end
$a' \leftarrow a[i \mapsto v]$;
return (a', l)

[a] attempt to evaluate the index to a literal using the SMT solver and two fresh variables id_1 and id_2
[b] check if index is always smaller than length
[c] an out-of-bound access can occur on infeasible paths, therefore, a feasibility check is required before being able to return an invalid verdict

Algorithm 2: access array

Input: Array tuple (a, l), path constraints pc, and index expression i
Output: Expression at index i, infeasible or a verdict

$pc_\wedge \leftarrow conjunct(pc)$;
$m \leftarrow satisfiable(pc_\wedge \wedge i = id_1)^a$;
if $m \neq \bot$ **then**
  if $satisfiable(pc_\wedge \wedge i = id_2 \wedge i \neq m(id_1)) = \bot$ **then**
    $i \leftarrow m(id_1)$;
  end
end
if $satisfiable(pc_\wedge \implies i \geq l)^b$ **then**
  if $satisfiable(pc_\wedge)^c$ **then**
    return *Invalid*;
  else
    return *Infeasible*;
  end
end
return $a(i)$

[a] attempt to evaluate the index to a literal using the SMT solver and two fresh variables id_1 and id_2
[b] check if index is always smaller than length
[c] an out-of-bound insertion can occur on infeasible paths, therefore, a feasibility check is required before being able to return an invalid verdict

Object Objects are represented by the mapping o, which maps identifiers to expressions. An object is initialized lazily if it is an argument to the `Main.main` method. If an object is initialized during symbolic execution, all of its fields adopt their default values: `false` for booleans, `0` for integers, and `null` for references.

For instance, consider a linked list `class LL { -Bool v; -LL next; }`. If an instance of this class is initialized as an argument of the `Main.main` method, the field `v` will be initialized to a symbolic boolean and the field `next` to a symbolic reference. In contrast, if an object of class `LL` is initialized during the symbolic execution, the initial values of fields `v` and `next` are `false` and `null`, respectively.

We define two operations on objects $update_{obj}$ which updates the value that a field maps to and $access_{obj}$ returning the value of a field according to the initialization rules outlined in the previous paragraph.

3.3.7 Stack

The stack of an execution path is represented by a stack of frames (s, m), consisting of a scope s and a mapping m from identifiers to expressions. Four operations are defined on the stack (1) $push_{stack}$, which pushes a new frame on the stack; (2) pop_{stack}, removing the top-most frame; (3) $insert_{stack}$, inserting a variable in the top-most frame; and (4) $access_{stack}$, get top-most variable from stack.

3.3.8 Heap

The heap of an execution path is implemented as a mapping h, from concrete references to reference values. We define two operations on the heap $insert_{heap}$ and $access_{heap}$, which are outlined in algorithms 3 and 4.

Algorithm 3: insert Heap

Input: Heap h, the symbolic reference *symref*, and the reference value to be inserted, v
Output: Updated heap

```
switch symref do
    case Concrete(r) do
        return h[r ↦ v];
    end
    case Symbolic(l, r, rv) do
        v ← h(r);
        if v ≠ ⊥[a] then
            return v;
        else
            if satisfiable(pc_∧ ⟹ l = null)[b] then
                if satisfiable(pc_∧)[c] then
                    return Invalid;
                else
                    return Infeasible;
                end
            else
                return h[r ↦ v];
            end
        end
    end
end
```

[a] a reference can only be inserted into the heap if it can never be null, therefore, if a reference is already in the heap, a null check is not required

[b] ensure that the symbolic reference can never be null

[c] inserting with a null reference can occur on infeasible paths, therefore, a feasibility check is required before being able to return an invalid verdict

Algorithm 4: access Heap

Input: Heap h, path constraints pc, the symbolic reference as expression $symref$
Output: Tuple of the updated heap and reference value, infeasible or a verdict

$pc_\wedge \leftarrow conjunct(pc)$;
switch $symref$ **do**
  case *Concrete(r)* **do**
    return $(h, h(r))$;
  end
  case *Symbolic(l, r, rv)* **do**
    $v \leftarrow h(r)$;
    if $v \neq \bot^a$ **then**
      return (h, v);
    else
      if $satisfiable(pc_\wedge \implies l = \texttt{null})^b$ **then**
        if $satisfiable(pc_\wedge)^c$ **then**
          return *Invalid*;
        else
          return *Infeasible*;
        end
      else
        return $(h[r \mapsto rv], rv)^d$;
      end
    end
  end
end

[a] a reference can only be inserted into the heap if it can never be null, therefore, if a reference is already in the heap, a null check is not required

[b] ensure that the symbolic reference can never be null

[c] accessing a null reference can occur on infeasible paths, therefore, a feasibility check is required before being able to return an invalid verdict

[d] if the reference is checked we insert and return the symbolic reference's placeholder value rv

3.3.9 Algorithm

The symbolic execution is a Breath-First Search (BFS) through the CFG we generated in section 3.2. During this search, we keep track of the state of each distinct execution path. The

Definition 3.3.1 (Symbolic State)**.** The Symbolic State (SS) is a triple (s, h, pc) where

- s is the stack
- h is the heap
- pc is the set of path constraints

The algorithm driving the symbolic execution can be seen in listing 5, we elaborate on the updating of the state in subsection 3.3.10.

Algorithm 5: The symbolic execution algorithm

Input: Starting node *node* and maximum depth d_{max}
Output: Verdict

```
completely_verified ← True;
q ← queue with elements (initial_state, node, 0)
while q is not empty do
    (state, node, depth) ← dequeue q;
    if depth ≥ d_max then
        completely_verified ← False;
        continue
    end
    switch update_state(state, node) do
        case Infeasible do
            continue
        end
        case Invalid do
            return Invalid
        end
        case state' do
            for next_node adjacent to node do
                enqueue (state', next_node, depth + 1) in q;
            end
        end
    end
end
if completely_verified then
    return Valid
end
return Unknown
```

3.3.10 Updating Symbolic State

Updating the state is defined as function

$$update\_state : \text{SS} \times Node \rightarrow Infeasible \mid Verdict \mid \text{SS}$$

In the following eight paragraphs, we will elaborate on how function *update_state* updates the state for the eight different cases of nodes the SEE can encounter.

Assignment Statement. During an assignment, the left-hand side can be either an identifier, a field access or an array access. The right-hand side can be either an expression, an object field, an array access or an array initialization[2]. The assignment consists of two steps, evaluating the right hand side value to an expression and assigning said value to the memory location the left-hand side points to.

Evaluating the right-hand side of an expression is defined as the function $eval_{rhs}$, outlined in 3.4. This function maps the symbolic state and the right-hand side to either an expression, infeasible, or a verdict.

$$\begin{aligned}
eval_{rhs}(s, h, pc, rhs_{expr}(e)) &= substitute(s, e) \\
eval_{rhs}(s, h, pc, rhs_{init}(ty, len)) &= init_{arr}(ty, len) \\
eval_{rhs}(s, h, pc, rhs_{field}(obj, f)) &= \begin{cases} access_{obj}(ref_n, f) & \textbf{if } ref_n = stack_{access}(obj) \\ Invalid & \textbf{otherwise} \end{cases} \\
eval_{rhs}(s, h, pc, rhs_{index}(arr, i)) &= \begin{cases} access_{arr}(ref_n, i) & \textbf{if } ref_n = stack_{access}(obj) \\ Invalid & \textbf{otherwise} \end{cases}
\end{aligned} \tag{3.4}$$

Assigning an expression to the left-hand side is defined as the function *assign*, outlined in 3.5. This function maps the symbolic state and the to-be assigned expression to the

[2]Method and constructor invocations on the right-hand side are deconstructed as discussed in paragraph **Procedure Call** in subsection 3.2.1.

updated symbolic state, infeasible or a verdict.

$$
\begin{aligned}
assign(s,h,pc,lhs_{id}(id),e) &= (stack_{insert}(id,e),h) \\
assign(s,h,pc,lhs_{field}(obj,f),e) &= \begin{cases} (s,update_{heap}(h,r,rv'),pc) & \textbf{if } r = stack_{access}(obj) \\ & \textbf{and } rv = heap_{access}(r) \\ & \textbf{and } rv' = update_{obj}(rv,f,e) \\ Invalid & \textbf{otherwise} \end{cases} \\
assign(s,h,pc,lhs_{index}(arr,i),e) &= \begin{cases} (s,update_{heap}(h,r,rv'),pc) & \textbf{if } r = stack_{access}(obj) \\ & \textbf{and } rv = heap_{access}(r) \\ & \textbf{and } rv' = update_{arr}(rv,pc,i,e) \\ Invalid & \textbf{otherwise} \end{cases}
\end{aligned}
\tag{3.5}
$$

Finally, we define the function *update_state* as $assign(s,h,pc,lhs,eval_{rhs}(rhs))$. It is important to note that when any auxiliary function, such as $heap_{access}$, returns an infeasible or invalid result, this is propagated as the result of *assign* and *update_state*.

Assert Statement. Let pc' represent the list of path constraints with the appended assertion, and let *assert* denote the function that updates the SS.

$$
assert(s,h,pc) = \begin{cases} (s,h,pc') & \textbf{if } satisfiable(\neg combine(pc')) = \bot \\ Invalid & \textbf{otherwise} \end{cases}
$$

Assume Statement. Let pc' represent the list of path constraints with the appended assumption, and let *assume* denote the function that updates the SS.

$$
assume(s,h,pc) = (s,h,pc')
$$

Return Statement. Let e be the to-be returned expression, and let *return* denote the function that updates the SS, this function maps e to the special `retval` keyword in the stack.

$$return(s, h, pc) = (insert_{stack}(s, \texttt{retval}, e), h, pc)$$

Entering Procedures. The SEE categorizes methods, static methods, and constructors as procedures. Upon entering a procedure, the SEE can execute four actions:

1. **Push frame** to the stack using $push_{stack}$
2. **Assign arguments** their corresponding parameters
3. **Initialize object** on the heap and assign its reference to the `this` keyword
4. **assign** `this` the object that a non-static method is called on

The combination of actions performed depends on the types of procedures that the SEE encounters. This is outlined in Table 3.1. Actions are executed from left to right, as shown in the table.

Procedure	Push frame	Assign arguments	Initialize object	Assign `this`
Static method	X	X		
Method	X	X		X
Constructor	X	X	X	

TABLE 3.1: Actions performed when entering a procedure

Leaving Procedures. Upon leaving a procedure, the SEE can execute three actions:

1. **Release** `retval` from its scope by inserting it in the underlying frame, making it available in the scope the method returns to.

2. **Release** `this` from its scope by inserting it in the underlying frame, making it available in the scope the method returns to.

3. **Pop frame** from the stack using pop_{stack}

The combination of actions performed depends on the types of procedures that the SEE encounters. This is outlined in Table 3.2. Actions are executed from left to right, as shown in the table.

Procedure	Release `retval`	Release `this`	Pop frame
Static method	X		X
Method	X		X
Constructor		X	

TABLE 3.2: Actions performed when entering a procedure

It should be noted that, for simplicity, the SEE will always attempt to release `retval`, regardless of whether a method returns an expression. Execution will continue if no expression is found to be released.

Enter Main. Entering the `Main.main` method is the initialization of the symbolic execution. When the SEE encounters this node two actions are taken: the initial scope is pushed and all the parameters of `Main.main` are assigned a symbolic value.

Otherwise. Let *update_state* be the identity function applied to the passed SS.

3.4 Heuristics

In this section we will introduce five categories of heuristics that can increase performance in various parts of the SEE algorithm discussed in subsection 3.3.9. For each heuristic, we will include a preface indicating in which part(s) of the symbolic execution algorithm this heuristic is implemented.

3.4.1 Pruning

This heuristic has been implemented in the 'assume' case of the *update_state* function, extending the potential result to include infeasibility

Pruning, in this context, is attempting to remove infeasible paths using the SMT solver. Pruning proves efficient during execution of programs with many infeasible paths, considering that one infeasible path can spawn many equally infeasible branches. Each of these branches must be explored. In contrast, if a large majority of paths is feasible pruning can increase the amount of SMT solver invocations manyfold, which can outweigh the cost of exploring the infeasible paths.

We define pruning heuristics using a tuple (p, a) where $0.05 \leq p \leq 1$ and $0 \leq a \leq 1$. In this tuple, p is the probability of Jip attempting to prune a path, and a represents the rate at which we update p after each pruning attempt. For example, if $(p, a) = (0.5, 0.05)$, there is a 50 percent chance Jip decides to prune, and p is increased or decreased by 0.05 depending on whether the pruning attempt was successful or unsuccessful, respectively. We refer to algorithm 6 to further clarify this.

Using tuple (p, a) we define 3 types of pruning: (1) **Constant pruning** or $(p, a) = (1, 0)$; (2) **Probabilistic Pruning** or $(p, a) = (0.25, 0)$; and (3) **Adaptive Pruning** or $(p, a) = (0.5, 0.05)$

Algorithm 6: Pruning

Input: Tuple (p, a) and path constraints pc
Output: *bool* indicating whether path is pruned
$r \leftarrow$ a random number between 0 and 1;
if $p \leq r$ **then**
    if *isFeasible*(pc) **then**
        $p \leftarrow min(0.05, p - a)$;
        return *true*;
    else
        $p \leftarrow max(1, p + a)$;
        return *false*;
    end
else
    return *true*;
end

3.4.2 Interval Inference

This heuristic introduces an interval map to the symbolic execution algorithm. The interval map is updated every time the path constraints are extended in the 'assume' and 'assert' cases of the *update_state* function.

Interval Inference determines the interval of values a symbolic variable v can have. For example: in expression $3 < v < 8$ we can infer $v \in \{4, 5, 6, 7\}$ and in expression $x == 3v \wedge 3 < v < 8$ we can infer $x \in \{12, .., 21\}$. In this section we will present an interval inference function $infer_m$ and an iterative algorithm using $infer_m$. This information allows the Extended Expression Evaluator (see section 3.4.3) to evaluate more expressions, preventing the use of the computationally expensive SMT solver.

3.4.2.1 Inferring Interval

Intervals are defined using the 'extended interval' approach outlined by Hickey et al.[12], we altered the approach to support integers instead of reals. The set of integers is extended with $-\infty$ and $+\infty$, that is $\mathbb{Z}_e = \mathbb{Z} \cup \{-\infty, +\infty\}$, to support the case of an integer having no lower and/or upper boundary. Let an interval be noted as $\langle a, b \rangle$ which is the set $\{x \in \mathbb{Z}_e : a \leq x \leq b\}$. Let (3.6) be the addition, subtraction and multiplication functions adapted from Hickey et al.[12].

$$\begin{aligned} \langle a,b\rangle + \langle c,d\rangle &= \langle a+c, b+d\rangle \\ \langle a,b\rangle - \langle c,d\rangle &= \langle a-c, b-d\rangle \\ \langle a,b\rangle * \langle c,d\rangle &= \langle min(\{a*c, a*d, b*c, b*d\}), max(\{a*c, a*d, b*c, b*d\})\rangle \end{aligned} \tag{3.6}$$

Let inferring an interval be defined as

$$infer_i : (Expression \times IntervalMap) \rightarrow Interval$$

In the following definitions, let $I = infer_i(E, \mathcal{I})$, $I_1 = infer_i(E_1, \mathcal{I})$ and so on.

$$\begin{aligned} infer_i(lit(z),\ \mathcal{I}) &= \langle z, z\rangle \\ infer_i(var(i),\ \mathcal{I}) &= \mathcal{I}(i) \\ infer_i(unop(E, -),\ \mathcal{I}) &= \langle -1, -1\rangle * I \\ infer_i(binop(E_1, +, E_2),\ \mathcal{I}) &= I_1 + I_2 \\ infer_i(binop(E_1, -, E_2),\ \mathcal{I}) &= I_1 - I_2 \\ infer_i(binop(E_1, *, E_2),\ \mathcal{I}) &= I_1 * I_2 \\ infer_i(E,\ \mathcal{I}) &= \langle -\infty, +\infty\rangle \end{aligned}$$

3.4.2.2 Inferring Interval Map

Mappings of symbolic variables and their accompanying intervals are denoted as $\mathcal{I}$. Where $\mathcal{I}[x \mapsto i]$ returns a new mapping with $x \mapsto i$ inserted, $\mathcal{I}(x)$ returns the current interval i that x is mapped to or $\langle -\infty, +\infty \rangle$ if no interval is mapped to x, and $vars(\mathcal{I})$ returns the set of symbolic variables in $\mathcal{I}$ that are mapped to an interval.

Let $broaden_i$ and $broaden_m$ calculate the most pessimistic[3] interval(s) for two intervals or two interval maps respectively.

$$\begin{aligned} broaden_i(\langle a, b \rangle, \langle c, d \rangle) &= \langle min(a, c),\ max(b, d) \rangle \\ broaden_m(\mathcal{I}_a, \mathcal{I}_b) &= \sum_{x \in vars(\mathcal{I}_a) \cup vars(\mathcal{I}_b)} x \mapsto broaden_i(\mathcal{I}_a(x),\ \mathcal{I}_b(x)) \end{aligned}$$

Let $narrow_i$ and $narrow_m$ calculate the most optimistic interval(s) for two intervals or two interval maps respectively.

$$\begin{aligned} narrow_i(\langle a, b \rangle, \langle c, d \rangle) &= \begin{cases} \langle max(a, c),\ min(b, d) \rangle & \textbf{if } \langle a, b \rangle \cup \langle c, d \rangle \neq \emptyset \\ \langle -\infty, +\infty \rangle & \textbf{otherwise} \end{cases} \\ narrow_m(\mathcal{I}_a, \mathcal{I}_b) &= \sum_{x \in vars(\mathcal{I}_a) \cup vars(\mathcal{I}_b)} x \mapsto narrow_i(\mathcal{I}_a(x),\ \mathcal{I}_b(x)) \end{aligned}$$

The function that infers the interval mapping from an expression is defined as

$$infer_m : (Expression \times IntervalMap) \rightarrow IntervalMap$$

In the following definitions, let $I = infer_i(E, \mathcal{I})$, $I_1 = infer_i(E_1, \mathcal{I})$ and so on.

[3] Pessimistic in terms of how much information we infer about a symbolic variable

Logical Operators For disjunction, we must consider the worst-case scenario when joining the inferred intervals from both the left-hand and right-hand expressions. In the case of conjunction, we do the opposite.

$$\begin{aligned} \mathit{infer}_m(\mathit{binop}(E_1, ||, E_2),\ \mathcal{I}) &= \mathit{broaden}_m(\mathit{infer}_m(E_1,\ \mathcal{I}),\ \mathit{infer}_m(E_2,\ \mathcal{I})) \\ \mathit{infer}_m(\mathit{binop}(E_1, \&\&, E_2),\ \mathcal{I}) &= \mathit{narrow}_m(\mathit{infer}_m(E_1,\ \mathcal{I}),\ \mathit{infer}_m(E_2,\ \mathcal{I})) \end{aligned}$$

Equality Operators For equality we narrow the already inferred intervals from both the left-hand and right-hand expressions. Depending on whether one or both sides are variables, we map the narrowed interval to the left-hand, the right-hand, or both.

$$\mathit{infer}_m(\mathit{binop}(E_1, ==, E_2),\ \mathcal{I}) = \begin{cases} \mathcal{I}[x_1 \mapsto i][x_2 \mapsto i] & \textbf{if } E_1 = \mathit{var}(x_1) \textbf{ and } E_2 = \mathit{var}(x_2) \\ & \textbf{where } i = \mathit{narrow}_i(I_1, I_2) \\ \mathcal{I}[x \mapsto i] & \textbf{if } E_2 = \mathit{var}(x) \\ & \textbf{where } i = \mathit{narrow}_i(\mathcal{I}(x), I_1) \\ \mathcal{I}[x \mapsto i] & \textbf{if } E_1 = \mathit{var}(x) \\ & \textbf{where } i = \mathit{narrow}_i(\mathcal{I}(x), I_2) \\ \mathcal{I} & \textbf{otherwise} \end{cases}$$

Negated Comparison Operators For negated comparison operators we rewrite the expression and infer recursively.

$$\begin{aligned} \mathit{infer}_m(\mathit{unop}(!,\ \mathit{binop}(E_1, <, E_2)), \mathcal{I}) &= \mathit{infer}_m(\mathit{binop}(E_1, >=, E_2)) \\ \mathit{infer}_m(\mathit{unop}(!,\ \mathit{binop}(E_1, <=, E_2)), \mathcal{I}) &= \mathit{infer}_m(\mathit{binop}(E_1, <, E_2)) \\ \mathit{infer}_m(\mathit{unop}(!,\ \mathit{binop}(E_1, >, E_2)), \mathcal{I}) &= \mathit{infer}_m(\mathit{binop}(E_1, <=, E_2)) \\ \mathit{infer}_m(\mathit{unop}(!,\ \mathit{binop}(E_1, >=, E_2)), \mathcal{I}) &= \mathit{infer}_m(\mathit{binop}(E_1, <, E_2)) \end{aligned}$$

Comparison Operators For comparison we attempt to narrow the boundaries of the intervals on the left-hand and right-hand sides based on the encountered comparison

operator. For instance, given $x > 5$, the result would be $x = \mathit{narrow}_i(\mathcal{I}(x), \langle 6, +\infty \rangle)$.

$$\mathit{infer}_m(\mathit{binop}(E_1, <, E_2),\ \mathcal{I}) = \begin{cases} \mathcal{I}[x \mapsto i] & \textbf{if } E_1 = \mathit{var}(x_1) \textbf{ and } I_2 = \langle a, b \rangle \\ & \textbf{and } a > -\infty \\ & \textbf{where } i = \mathit{narrow}_i(\mathcal{I}, \langle -\infty, a - 1 \rangle) \\ \mathcal{I}[x \mapsto i] & \textbf{if } I_1 = \langle a, b \rangle \textbf{ and } E_2 = \mathit{var}(x_2) \\ & \textbf{and } b < +\infty \\ & \textbf{where } i = \mathit{narrow}_i(\langle b + 1, +\infty \rangle, \mathcal{I}(x_2)) \\ \mathcal{I} & \textbf{otherwise} \end{cases}$$

$$\mathit{infer}_m(\mathit{binop}(E_1, <=, E_2),\ \mathcal{I}) = \begin{cases} \mathcal{I}[x \mapsto i] & \textbf{if } E_1 = \mathit{var}(x_1) \textbf{ and } I_2 = \langle a, b \rangle \\ & \textbf{and } a > -\infty \\ & \textbf{and } i = \mathit{narrow}_i(\mathcal{I}(x_1), \langle -\infty, a \rangle) \\ \mathcal{I}[x \mapsto i] & \textbf{if } I_1 = \langle a, b \rangle \textbf{ and } E_2 = \mathit{var}(x_2) \\ & \textbf{and } b < +\infty \\ & \textbf{where } i = \mathit{narrow}_i(\langle b, +\infty \rangle, \mathcal{I}(x_2)) \\ \mathcal{I} & \textbf{otherwise} \end{cases}$$

$$\mathit{infer}_m(\mathit{binop}(E_1, >, E_2),\ \mathcal{I}) = \begin{cases} \mathcal{I}[x \mapsto i] & \textbf{if } E_1 = \mathit{var}(x_1) \textbf{ and } I_2 = \langle a, b \rangle \\ & \textbf{and } a > -\infty \\ & \textbf{and } i = \mathit{narrow}_i(\mathcal{I}(x_1), \langle b + 1, +\infty \rangle) \\ \mathcal{I}[x \mapsto i] & \textbf{if } I_1 = \langle a, b \rangle \textbf{ and } E_2 = \mathit{var}(x_2) \\ & \textbf{and } b < +\infty \\ & \textbf{where } i = \mathit{narrow}_i(\langle -\infty, a - 1 \rangle, \mathcal{I}(x_2)) \\ \mathcal{I} & \textbf{otherwise} \end{cases}$$

$$\mathit{infer}_m(\mathit{binop}(E_1, =>, E_2),\ \mathcal{I}) = \begin{cases} \mathcal{I}[x \mapsto i] & \textbf{if } E_1 = \mathit{var}(x_1) \textbf{ and } I_2 = \langle a, b \rangle \\ & \textbf{and } a > -\infty \\ & \textbf{and } i = \mathit{narrow}_i(\mathcal{I}(x_1), \langle b, +\infty \rangle) \\ \mathcal{I}[x \mapsto i] & \textbf{if } I_1 = \langle a, b \rangle \textbf{ and } E_2 = \mathit{var}(x_2) \\ & \textbf{and } b < +\infty \\ & \textbf{where } i = \mathit{narrow}_i(\langle -\infty, a \rangle, \mathcal{I}(x_2)) \\ \mathcal{I} & \textbf{otherwise} \end{cases}$$

$$\mathit{infer}_m(E,\ \mathcal{I}) = \mathcal{I}$$

3.4.2.3 Iterative Interval inference

In the case of inferring x and v in $x == 3v \wedge 3 < v < 8$, function $infer_m$ would need the information that the interval of v is $\langle 4, 7\rangle$ to narrow the interval of x to $\langle 12, 21\rangle$, this information is only available after the first evaluation of $infer_m$. To calculate the intervals more accurate we define the iterative inference algorithm 7, parameterizing over the maximum number of iterations i.

Algorithm 7: Iterative Inference

Input: an expression e and the max. inference depth i
Output: *IntervalMap*
$\mathcal{I} \leftarrow [\,]$;
$\mathcal{I}' \leftarrow infer_m(e, \mathcal{I})$;
while $i > 0 \wedge \mathcal{I} \neq \mathcal{I}'$ **do**
  $\mathcal{I} \leftarrow \mathcal{I}'$;
  $\mathcal{I}' \leftarrow infer_m(e, \mathcal{I}')$;
  $i \leftarrow i - 1$;
end
return $\mathcal{I}'$;

Note that the iterative inference algorithm will always terminate, as shown in the proof below. However, since we assume the information gained from each successive iteration diminishes, we implemented a bound on the number of iterations.

Proof. Assume we have an integer variable x and the most precise interval that can be estimated for this variable is $\langle a^\top, b^\top\rangle$. We now consider any arbitrary interval $\langle a, b\rangle$ such that $a \leq a^\top$ and $b \geq b^\top$. In this case, the set difference $\{a, .., b\} \setminus \{a^\top, .., b^\top\}$ is a finite set, there are only a finite number of integers in the interval $\langle a, b\rangle$ that are not in the interval $\langle a^\top, b^\top\rangle$. Therefore, interval $\langle a, b\rangle$ can be narrowed to $\langle a^\top, b^\top\rangle$ in a finite amount of steps.

Moreover, the number of integer variables in an expression is finite, and each iteration of our iterative inference algorithm narrows down at least one interval (otherwise it terminates), thus the algorithm always terminates. □

3.4.3 Extended Expression Evaluation

This heuristic is applied when saving expressions in the stack and heap. Furthermore, the SEE will preface each invocation of *satisfiable*(e), the function interfacing with the SMT solver, with an attempt to evaluate e using this heuristic. If e is evaluated to a literal, the invocation of *satisfiable* is skipped.

Expression Evaluation, designed by Koppier[15], aims to simplify an expression $e \in$ *Expression* by evaluating it where possible. We extend Expression Evaluation with several simplification rules and incorporate the mapping of inferred intervals from section 3.4.2. The benefit of Expression Evaluation is twofold: by evaluating expressions with this method we prevent the use of our expensive SMT solver and we reduce memory usage by minimizing the size of expressions.

Let the Expression Evaluation function designed by Koppier[15] be *eval*, then we extend *eval* as follows:

$$eval : (Expression \times IntervalMap) \rightarrow Expression$$

In the following definitions, let $E' = eval(E, \mathcal{I})$, $E'_1 = eval(E_1, \mathcal{I})$ and so on.

Double Unary Operators.

$$eval(unop(unop(E, !), !), \mathcal{I}) = eval(E, \mathcal{I})$$
$$eval(unop(unop(E, -), -), \mathcal{I}) = eval(E, \mathcal{I})$$

Duplicate Variable Comparison.

$$eval(binop(E_1, ==, E_2), \mathcal{I}) = lit(\texttt{true}) \quad \textbf{if } E'_1 = var(x) \textbf{ and } E'_2 = var(x)$$

Interval Point.

$$eval(E,\ \mathcal{I}) = lit(a) \quad \textbf{if } infer_i(E', \mathcal{I}) = \langle a, a \rangle$$

Interval Equality. Function *eval* defines inequality in terms of equality e.g. $x! = y$ becomes $!(x == y)$.

$$eval(binop(E_1, ==, E_2),\ \mathcal{I}) = \begin{cases} lit(\texttt{false}) & \textbf{if } infer_i(E_1', \mathcal{I}) = \langle a, b \rangle \textbf{ and } infer_i(E_2', \mathcal{I}) = \langle c, d \rangle \\ & \textbf{and } \ \langle a, b \rangle \cap \langle c, d \rangle = \emptyset \end{cases}$$

Interval comparison. Function *eval* defines all comparisons in terms of $<$ e.g. $(x <= y) = !(y < x)$, $(x > y) = (y < x)$ and $x >= y = !(x < y)$.

$$eval(binop(E_1, <, E_2),\ \mathcal{I}) = \begin{cases} lit(\texttt{true}) & \textbf{if } infer_i(E_1', \mathcal{I}) = \langle a, b \rangle \textbf{ and } infer_i(E_2', \mathcal{I}) = \langle c, d \rangle \\ & \textbf{and } \ b < c \\ lit(\texttt{false}) & \textbf{if } infer_i(E_1', \mathcal{I}) = \langle a, b \rangle \textbf{ and } infer_i(E_2', \mathcal{I}) = \langle c, d \rangle \\ & \textbf{and } \ d < a \end{cases}$$

Dynamic quantifiers. Let $\odot$ be a *forall* or *exists* quantifier, then *eval* will evaluate arrays where the concrete size can be inferred.

$$eval(\odot(a, i, v, E),\ \mathcal{I}) = \begin{cases} eval(\odot(a, i, v, i_3, E),\ \mathcal{I}) & \textbf{if } infer_i(sizeof(a), \mathcal{I}) = \langle a, b \rangle \\ & \textbf{and } b < +\infty \\ \odot(a, i_2, i_3, E) & \textbf{otherwise} \end{cases}$$

3.4.4 Expression Caching

This heuristic introduces an expression cache to the symbolic execution algorithm. The expression cache is consulted prior to each invocation of *satisfiable*, the function interfacing with the SMT solver, and the cache is updated following each invocation of *satisfiable*.

Expression Caching is a popular optimization technique[27][19] that aims to reduce the usage of the SMT solver by memoizing previously solved expressions. We have implemented both Expression caching as described by Koppier[15] and Normalized Expression Caching, which will be explained in section 3.4.4.1.

3.4.4.1 Normalized Expression Caching

When consulting the expression cache we would prefer that the expression cache recognizes expression equivalence e.g. $-1+2=2-1$ or $a \wedge b = b \wedge a$. Therefore we propose Normalized Expression Caching, a heuristic that uses two methods to increase the chance of equivalent expression discovery: (1) apply several normalizations e.g. $1-1$ becomes $1+(-1)$; and (2) recursively collect and sort all operands of commutative operators e.g. to let $1+2+3$ equal $3+2+1$.

Let *collect* be a function to recursively collect operands of the same operation.

$$\begin{aligned} collect(\otimes, binop(E_1, \otimes, E2)) &= collect(\otimes, E_1) + + collect(\otimes, E_2) \\ collect(\otimes, E) &= [E] \end{aligned}$$

Let $hash^4$ be a function that hashes any value to a 32-bit integer, and let $sort_{list}$ be a sort function implemented using merge-sort. Then we define hash function of *Expression* as

$hash_{expr} \circ eval$

$$
\begin{aligned}
sort_{op} &= sort_{list} \circ (map\ hash_{expr}) \circ collect \\
hash_{expr}(binop(E_1, -, E_2)) &= hash_{expr}((binop(unop(-, E_1), +, unop(-, E_2))) \\
hash_{expr}(binop(E_1, +, E_2)) &= hash((+, sort_{op}(E_1, +, E_2))) \\
hash_{expr}(binop(E_1, *, E_2)) &= hash((*, sort_{op}(E_1, *, E_2))) \\
hash_{expr}(binop(E_1, \&\&, E_2)) &= hash((\&\&, sort_{op}(E_1, \&\&, E_2))) \\
hash_{expr}(binop(E_1, ||, E_2)) &= hash((||, sort_{op}(E_1, ||, E_2))) \\
hash_{expr}(E) &= hash(E)
\end{aligned}
$$

Given that $sort_{list}$ implements merge sort, the complexity of $hash_{expr}(e)$ is $\mathcal{O}(n\ log\ n)$, where n is the number of subexpressions e consists of.

3.4.5 Parallel Solving

This heuristic extends the *satisfiable* function.

Parallel solving, as suggested by Palikareva et al. [19], aims to speed up constraint solving by simultaneously running different SMT solvers. One thread is spawned for each type of SMT solver supported by Jip[5], and whenever the first thread returns a solution the SEE resumes symbolic execution. This approach allows the SEE to utilize the best suited SMT solver for each constraint, at the cost of two extra CPU cores.

[4]The function *hash* is akin to Rust's `derive(Hash)` macro
[5]Currently these are Z3[3], CVC4[1] and Yices2[2]

Chapter 4

Results

In this Chapter we will reflect on the performance, soundness, and completeness of Jip. We will use a diverse set of verification tasks from the Competition on Software Verification to give an insight into the performance of Jip with and without the various heuristics introduced in Chapter 3. We will compare the performance of Jip with seven other verification tools[1]. Concluding this section we will substantiate the claims of completeness and soundness in Jip by comparing its verification result with the other verifiers. This Chapter aims to address the following questions:

1. What is an (approximate) optimal combination of heuristics?
2. How is Jip's performance compared to current state-of-the-art verifiers?
3. Is Jip compelete and sound?

4.1 Benchmark Setup

For the setup of the benchmarkss we have chosen to use the benchmarking tool BenchExec[6] and verification tasks provided by SV-COMP 2023[7], the largest competition for software

[1]The public repository containing the SEE can be found at `https://github.com/tjausm/Jip`, the results of the five benchmarks are located in the `benchmarks` folder

verification tools. Using these tools ensures the results of the benchmarks are reproducible and comparable with other verification tools.

Verification tasks In the context of SV-COMP, a verification task is a program with several lines of code added to verify the correctness of the program, one program can spawn many verification tasks e.g. an implementation of a RedBlack tree can have one task to verify the tree is always in balance and one task to verify an insertion never generates a null pointer exception. A task that contains a verification error yields an invalid verdict, all other tasks yield a valid verdict.

SV-COMP provides 23,805 verification tasks in C and 586 verification tasks in Java. For the benchmarks, we translated 81 Java verification tasks, which stemmed from 19 programs, into OOX. The Java dataset was chosen as the source because there were already several verification tasks from this dataset available in OOX. This dataset was divided into 12 folders, each containing verification tasks from a different source, such as the MinePump suite or the test suite of the JBMC verifier. We selected and translated three complete folders: `jayhorn-recursive`, `algorithms`, and `MinePump`. These folders were chosen because all the programs within them could be translated into OOX.

The translated verification tasks can be found in the public repository's `benchmarks/verification-tasks` folder[24].

Scoring Schema Throughout the following sections, we will utilize the scoring schema, provided by SV-COMP, outlined in Table 4.1, to assess the performance of Jip and the other tools.

Reported result	Points	Description
UNKNOWN	0	Failure to compute verification result
FALSE correct	+1	Violation of property in program was correctly found
FALSE incorrect	-16	Violation reported, but property holds (false alarm)
TRUE correct	+2	Program correctly reported to satisfy property
TRUE incorrect	-32	Incorrect program reported as correct

TABLE 4.1: Scoring schema for SV-COMP 2023[7]

BenchExec BenchExec[6] will be used to measure and compare the performance of Jip and other tools on the selected verification tasks. BenchExec is a tool developed for SV-COMP to control and measure the verification runs in the competition. It helps ensure reliable and reproducible competition results by providing a standardized environment for running benchmarks, collecting data, and comparing results. It ensures that the benchmarks are executed under the same conditions, with the same resources, and with the same inputs.

Quantile Function Graphs In the following sections we will use the quantile function several times to compare the benchmarking of verification tasks. This function calculates, for each data point (x, y), the minimum timeout y needed to accumulate the maximum amount of points x, according to the scoring schema defined in Table 4.1. Given the example benchmark in Table 4.2 the plotted points would be (1,5), (3, 10), (4, 20) and (6, 40). The first point represents only solving task one with a five-second timeout, the second point represents solving task one and two with a ten-second timeout, and so forth.

	task 1	task 2	task 3	task 4
time (s)	5	10	20	40
score	1	2	1	2

TABLE 4.2: Example benchmark results

Verification tools The performance of Jip will be compared to the performance of Java verifiers on the Java equivalent of the 81 OOX verification tasks. We have chosen to compare Jip with seven of the eight[2] Java verifiers that have competed in SV-COMP 2023, which are COASTAL[25], GDart[18], Java Ranger[23], JBMC[9], Jayhorn[13], MLB[17] and SPF[20].

Acronyms When necessary we will refer to the heuristics using the acronyms in Table 4.3. The SMT solvers will be referred using their full names: CVC4, Yices2, and Z3. Combinations of heuristics will be denoted as "EEE+CP" or "II+EC+CVC4", if no SMT solver is specified Z3 is used.

Heuristic	Acronym
Extended Expression Evaluator	EEE
Interval Inferrer	II
Interval Inferrer (two iterations)	II2
Probabilistic Pruner	PP
Adaptive Pruner	AP
Constant Pruner	CP
Expression Cacher	EC
Normalized Expression Cacher	NEC
No Heuristics	NH

TABLE 4.3

4.2 Heuristics

Unless otherwise specified, all benchmarks in the Heuristics section are executed with a 20-second timeout, a 1,000MB memory limit, and a single CPU core.

[2]JDart was left out of this comparison due to installation problems

In this section, we evaluate the different configurations of heuristics that Jip can use, with the aim of approximating an optimal configuration of heuristics. Figure 4.1 provides a view of all 11 heuristics, divided into four distinct stages. The stages of the heuristics are arranged from left to right, reflecting the sequence in which they are used during symbolic execution. For example, when an `assume` statement is encountered, the SEE invokes the Extended Expression Evaluator (EEE). If this stage fails and the SEE chooses to prune, the Expression Cache is checked. Only if the expression is not cached does the SEE proceed to one of our four SMT solvers.

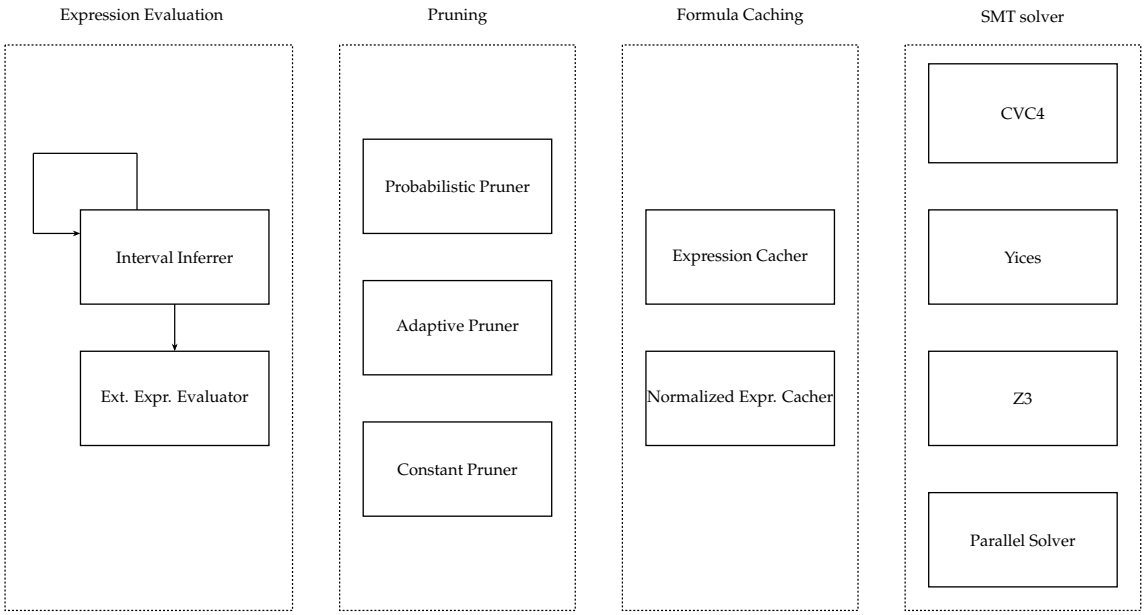

FIGURE 4.1: The heuristics pipeline

All heuristic stages are optional, barring the SMT solver. Further, all heuristics within a single stage are mutually exclusive, with the exception of II and EEE. In this case, II is dependent on EEE. Moreover, II can be configured to run an arbitrary number of iterations.

Given the constraints, and limiting II to two iterations, we have four choices for the Expression Evaluation stage, four for the Pruning stage[3], three for the Formula Caching stage, and four for the SMT solver stage. This results in a total of $4 \cdot 4 \cdot 3 \cdot 4 = 192$ heuristic

configurations.

Comparing all these configurations across our set of 81 verification tasks is infeasible. As such, we propose an incremental approach to approximate the optimal heuristic configuration. We start by comparing all Expression Evaluation heuristics. Subsequently, we compare all Pruning heuristics, using the best configuration from the Expression Evaluation stage, and proceed in this manner. During these comparisons, we will also discuss the performance and peculiarities of the individual heuristics.

4.2.1 Expression Evaluation

In the subsequent paragraphs, we will investigate the difference in performance between NH, EEE, II, and II2. The performance of these heuristic configurations across the 81 verification tasks is depicted in Figure 4.2. For further insight, a detailed benchmark of the four configurations in seven verification tasks[4] is presented in Tables 4.4, 4.5, and 4.6. This detailed benchmark uses a 100-second timeout, 12000 MB memory limit and a single CPU core.

[3]These include the three pruning heuristics and the option with no heuristics, the same applies for expression evaluation and formula caching

[4]The verification tasks used for the detailed benchmarks can be found in the `benchmarks/verification-tasks` folder in the public repository (https://github.com/tjausm/Jip)

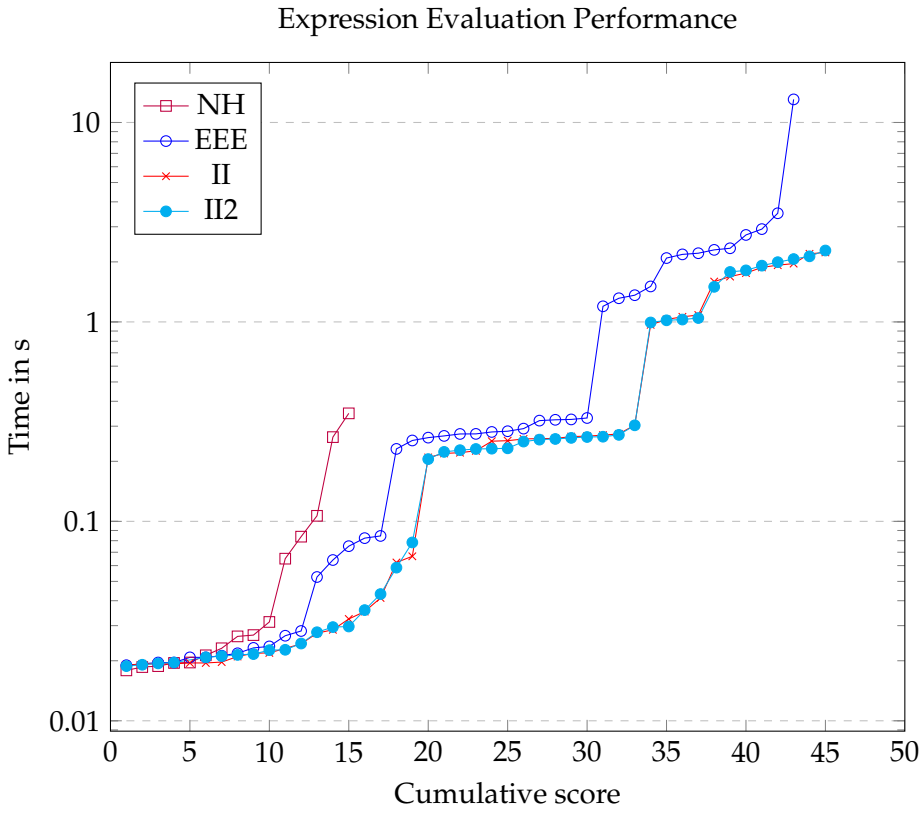

FIGURE 4.2: Quantile function (see 4.1) graph of the benchmarking results for the three Expression Evaluation heuristics

NH versus EEE The most notable difference between NH and EEE is the cumulative score, with NH and EEE scoring 16 and 44 points respectively. In most cases, NH exhausts the time or memory limit prior to completing the tasks. NH's lower score can be primarily attributed to two factors:

1. NH explores all paths, including infeasible ones. Looking in Tables 4.4 and 4.5 we see that NH explores 1000s of extra paths compared to the EEE or II while only

	time (s)	memory (MB)	depth	paths explored	SMT calls
NH	123.1	11530	54	738	5315
EE	13.1	1250	65	1508	11655
II	0.0406	38.1	65	0	21
II2	0.044	38.1	65	0	21

TABLE 4.4: Verification details of `BellmanFord-FunUnsat01` (red text indicates unfinished verification)

	time (s)	memory (MB)	depth	paths explored	SMT calls
NH	106.1	12553	156	2651	13194
EEE	3.51	172	223	0	316
II	2.23	138	223	0	2
II2	2.13	131	223	0	2

TABLE 4.5: Verification details of MinePump's `spec1-5_product15`(red text indicates unfinished verification)

reaching a fraction of the depth reached by the EEE and II.

2. NH does not reduce the size of the expressions, resulting in significantly higher memory usage which in turn decreases the verifier's performance. This is particularly noticeable in the `BellmanFord-FunUnsat01` task, as outlined in Table 4.4.

EEE versus II Figure 4.2 shows a significant increase in performance caused by II on the majority of tasks. Furthermore II scores 1 more point, by verifying task `MergeSortIterative-FunUnsat01`.

In Table 4.6 we see that II does not completely explore any paths in the `MergeSortIterative-FunUnsat01` task and only uses 215 SMT calls to find a violation of the assertion, whereas EEE explores 15727 paths and invokes the SMT solver 68131 times to reach one-third of the depth needed to find the bug. From this, we conclude that II can solve the task because it is able to prune all infeasible paths.

In the 32 minepump tasks, of which `spec1-5_product15` can be seen in Table 4.5, II also regularly increases performance with 10%-40%. We assume II does this by bringing down the amount of SMT calls needed to solve a problem, as evidenced in the `spec1-5_product15` task, where SMT solver calls were reduced from 316 to 2.

	time (s)	memory (MB)	depth	paths explored	SMT calls
NH	114.1	11644	64	6464	133926
EEE	108.1	9437	73	15727	68131
II	0.285	50.7	138	5	46
II2	0.266	50.7	138	5	3

TABLE 4.6: Verification details of `MergeSortIterative-FunUnsat01` (red text indicates unfinished verification)

II versus II2 The performance difference between II and II2 is not apparent in Figure 4.2. Only in five instances does II2 decrease the number of necessary SMT solver calls needed to verify a task. Among these, the most significant reduction is observed in the `MergeSortIterative-FunUnsat01` task, where SMT solver calls drop from 46 to 3. Interestingly, in two of those 5 cases, II performs worse than II2 despite reducing SMT solver calls.

Concluding Expression Evaluation We have gathered the total runtime of all four heuristics in Table 4.7. The data reveals that both II and II2 halve the runtime when compared to EEE. However, the difference between II and II2 in terms of runtime is minimal. In the following sections, we will continue using II because it has the smallest total verification time, albeit with a very minor difference in comparison to II2.

	Score	Total time
NH	15	1.12
EEE	44	43.1
II	45	23.5
II2	45	23.6

TABLE 4.7: Expression evaluation summary

4.2.2 Pruning

Considering the optimal configuration derived from the previous section, we have four heuristic configurations to compare: (1) II; (2) II+PP; (3) II+AP; and (4) II+CP. The performance of these configurations, as observed across 81 verification tasks, is presented in Table 4.10. The detailed data of II, II+PP, II+AP, and II+CP executing 2 verification tasks[5] during a repeated benchmark, using 12000 MB memory and a 100-second timeout, are presented in Tables 4.8 and 4.9. Subsequent sections will examine the performance differences among II, II+PP, II+AP, and II+CP, using Tables 4.8 and 4.9 to conclude with a comparison of the overall performance of II, II+PP, II+AP, and II+CP.

	time (s)	depth	paths pruned	avg. prune probability	SMT calls
II	106.1	69	164	0%	386
II+PP	95,95	104	14808	25%	15018
II+AP	0.906	104	299	81.3%	541
II+CP	0.831	104	5	100%	516

TABLE 4.8: Verification details of `UnSatFibonacci01`

	time (s)	depth	paths pruned	avg. prune probability	SMT calls
II	0.285	138	1606	0%	46
II+PP	0.393	138	1606	25%	443
II+AP	0.906	104	299	81.3%	541
II+CP	0.831	104	5	100%	516

TABLE 4.9: Verification details of `MergeSortIterativeFunUnsat01`

Best Case for Pruning Both II+AP and II+CP score an additional point by verifying the task `UnSatFibonacci01`. During a benchmark with a 100-second timeout, II+PP also scores the extra point. According to Table 4.8, II+CP only prunes five paths in the task `UnSatFibonacci01`, whereas II+PP prunes 14,808 paths. This suggests that each infeasible path generates a significant number of other infeasible paths, thereby rewarding heuristics that prune aggressively. The performance of II further supports this conclusion, as it only reaches a depth of 69 with a relatively low number of SMT calls. This suggests that II does not reach the depth needed to find the bug in the task, and instead is exploring countless infeasible paths until the timeout is reached.

[5]The verification tasks used for the detailed benchmarks can be found in the `benchmarks/verification-tasks` folder in the public repository (https://github.com/tjausm/Jip)

Worst Case for Pruning Contrastingly, in task `MergeSortIterative-FunUnsat01` the pruning heuristics lead to the largest decrease in performance, more than doubling the verification time in the case of II+CP. In this task, II manages to prune 1606 paths without the SMT solver, and none of the pruning heuristics manage to prune additional paths. In this instance, II+AP minimizes the pruning probability, limiting the performance decrease compared to II. But the increased amount of SMT calls, triggered by prune attempts, negatively impacts the performance of II+CP significantly.

Concluding Pruning We have gathered the total runtime of all four heuristic configurations in Table 4.10. Configuration II+PP is the fastest, but if we included the verification time of the task `UnSatFibonacci01` from the detailed benchmark in 4.8 this configuration would have a total time of 114.85. In the last two sections, we saw the task `UnSatFibonacci01` where pruning caused the biggest increase in performance and we saw `MergeSortIterative-FunUnsat01` where pruning halved the speed of the verifier. The adaptive pruning was supposed to find the middle ground, adjusting the prune probability based on the successfulness of pruning, but from Tables 4.8 and 4.9 we conclude that the more aggressive pruning in configuration II+CP outperforms II+AP. Consequently, in the following sections, we will continue using II+CP.

	Score	Total time
II	45	23.5
II+PP	45	18.9
II+AP	46	22.8
II+CP	46	21.3

TABLE 4.10: Pruning summary

4.2.3 Expression Caching

Taking into account the best-performing configuration from previous sections, we have three heuristic configurations: (1) II+CP; (2) II+CP+EC; and (3) II+CP+NEC. The performance of these configurations on the 81 verification tasks can be seen in Table 4.13. In Tables 4.12 and 4.11, we present detailed data[6] for the task where expression caching performs the worst, `UnsatFibonacci01`, and the task where it performs the best, `MergeSortIterative-FunUnsat01`, respectively. In subsequent sections we compare the best- and

worst case scenarios for Expression caching to conclude with a comparison of the overall performance of II+CP, II+CP+EC, and II+CP+NEC.

	time (s)	memory (MB)	cache hits	SMT calls
II+CP	0.656	50.9	0	1645
II+AP+EC	0.308	51	1525	120
II+CP+NEC	0.306	50.8	1525	120

TABLE 4.11: Verification details of `MergeSortIterative-FunUnsat01`

	time (s)	memory (MB)	cache hits	SMT calls
II+CP	0.831	60.6	0	516
II+AP+EC	1.04	126	1	515
II+CP+NEC	2.18	126	1	515

TABLE 4.12: Verification details of `UnSat-Fibonacci01`

Best Case for Expression Caching During the `MergeSortIterative-FunUnsat01` task both expression caching heuristics significantly improve performance, doubling the performance by caching 1525 of 1645 SMT calls that II+CP needs to verify the task. Interestingly, II+CP+NEC performs as well as II+CP+EC on this task. This is likely due to the majority of expressions in this task already being in a normalized form. This observation is further supported by the fact that normalization does not reduce the size of the expression cache; both heuristics utilize similar amounts of memory.

Worst Case for Expression Caching Contrastingly, for task `UnSatFibonacci01` in Table 4.12 we see that it takes II+CP 516 SMT calls to verify it, the expression caching configurations only manage to reduce the amount of SMT calls by one. The difference in performance decrease between the two expression caching heuristics is also significant, the cheaper II+CP+EC reduces performance by 0.21 seconds, or approximately 25 percent, whereas the more expensive II+CP+NEC reduces performance by 1.35 seconds, or approximately 110 percent. Additionally, a stark difference in memory usage between the expression caching heuristics, II+CP+NEC using 83.4 MB compared to II+CP+EC's usage of 126 MB. This difference in memory usage suggests that II+CP+NEC manages to significantly reduce the size of the expression cache through expression normalization.

[6]The verification tasks used for the detailed benchmarks can be found in the `benchmarks/verification-tasks` folder in the public repository (https://github.com/tjausm/Jip)

	Score	Total time
II+CP	46	21.3
II+CP+NEC	46	21.6
II+CP+EC	46	19.3

TABLE 4.13: Expression caching summary

Concluding Expression Caching In concluding the review of the expression caching heuristics we have gathered the total runtime of all three heuristic configurations in Table 4.13. There were no tasks in which the normalization managed to affect the amount of cache hits. Consequently, the total verification time for II+CP+NEC is equivalent to the sum of the normalization time and the verification time for II+CP+EC. Given that normalization only decreases performance we will proceed with II+CP+EC in the subsequent section.

4.2.4 SMT Solver

In section 4.2.4.1, we discuss how the initial benchmarks show the parallel solver heuristic (outlined in subsection 3.4.5) to be infeasible. Next, we compare the three supported SMT solvers in section 4.2.4.2.

4.2.4.1 Infeasability of Parallel Solving

To understand the problem with parallel solving we must dive deeper in how Jip uses its SMT solver. During symbolic execution Jip spawns a separate process running the SMT solver, whenever Jip wants to know the satisfiability of an expression it queries this separate process, halting symbolic execution until the SMT solver returns a result.

Parallel solving requires spawning three separate processes running three SMT solvers. All three SMT solvers are consulted for each query, resuming symbolic execution with the

result of the fastest solver. In subsequent SMT calls, we cannot guarantee that all three SMT solvers have solved their last problem, considering that Jip resumes execution with the result of the fastest SMT solver. Thus we are required to spawn three new processes for each SMT invocation, spawning these processes takes initialization time and can lead to spawning more processes than available CPU cores. Initial benchmarks showed parallel solving to be 2-3x times slower on several tasks, supporting our concerns.

4.2.4.2 SMT solver comparison

We have compared nine configurations of heuristics with Z3 as SMT solver and found that II+CP+EC was the best performing. The comparison of the three supported solvers Z3, CVC4, and Yices 2 will be done using the best configuration of heuristics we have found so far, resulting in the following configurations (1) II+CP+EC[7]; (2) II+CP+EC+CVC4; and (3) II+CP+EC+Yices2. The results of this benchmark can be seen in Table 4.14.

Yices2 consistently outperforms the other solvers in all tasks, except for four. Z3 follows as the second fastest in all but four tasks. CVC4, while slower overall, surpasses the other SMT solvers in the four smallest verification tasks[8]. Though CVC4 is generally slower, it outperforms the other SMT solvers on the four smallest verification tasks by at most 0.005 seconds. This difference can be caused by a variety of factors such as the CVC4 implementation in Jip initializing faster. However, this difference is too insignificant to further investigate.

	Score	Total time
II+CP+EC	46	19.3
II+CP+EC+CVC4	46	22.4
II+CP+EC+Yices2	46	17.5

TABLE 4.14: SMT solver summary

We have gathered the total verification time of all three heuristic configurations in Table 4.14. Yices2 increases verification speed by 10 percent in comparison to II+CP+EC, the fastest heuristic configuration we have found so far. We conclude the heuristic section by choosing II+CP+EC+Yices2 as our approximation of an optimal heuristic configuration.

[7]When no solver is referenced Z3 is used by default

[8]These tasks are solved by all three configurations within 0.04 seconds

4.3 Verifier Comparison

Unless otherwise specified, all benchmarks in the Verifier Comparison section are executed with a 300-second timeout, a 10,000MB memory limit, and a single CPU core.

In this section we will compare the performance of the Jip, using the optimal heuristic configuration II+CP+EC+Yices2 we found in the previous section, to seven state-of-the-art verifiers. The verifiers will be benchmarked on the Java source of the OOX verification tasks used to benchmark Jip. The results of this benchmark can be seen in Figure 4.3 and Table 4.16.

	JBMC	Jip	MLB	Java Ranger	SPF	COASTAL	GDart	JayHorn
UnsatFibonacci01	3.92 s	2456 s	3.87 s	8.46 s	2.76 s	54.9 s	10.4 s	11 s
spec1-5_product15	1.65 s	1.55 s	2.15 s	4.72 s	2.63 s	2.63 s	10.7 s	132 s
spec1-5_product62	2.3 s	229 s	5.56 s	157 s	2.73 s	U	U	T

TABLE 4.15: Verification time in seconds of the seven verifiers on three tasks (U = Unknown, T = Timeout)

Point Difference Looking at Table 4.16 we can see that Jip is the second best verifier in terms of score. Jip can solve all the tasks that JBMC can, with the exception of `UnsatFibonacci02`. Repeated benchmarks without a time limit (see Table 4.15) reveal that Jip can indeed solve `UnsatFibonacci02`, but it takes 2456 seconds in contrast to the 3.92 seconds that JBMC requires. The likely cause of this difference is that JBMC has a bound on verifying loops that was configured for optimal performance with the SV-comp verification tasks[16].

Time Difference Compared to JBMC and MLB, the best and third-best verifiers respectively, Jip spends 7-12 times as long to finish its 51 successful verification tasks. Only 2 percent of this time is spent on verifying the 44 invalid verification tasks (tasks containing

Verifier	Score	IF	IT	CT	CF	Total time (s)
JBMC	60	0	0	7	46	80.3
Jip	59	0	0	7	45	941
MLB	54	0	0	7	40	124
Java Ranger	52	0	0	6	40	1140
SPF	49	0	0	7	35	91.5
COASTAL	40	0	0	1	39	533
GDart	36	0	0	1	34	316
JayHorn	33	0	1	12	41	3760

TABLE 4.16: Summarized results of all verifiers sorted from highest to lowest score (IF = incorrect false, IT = incorrect true, CT = correct true, and CF = correct false)

a bug), the other 98 percent is spent on verifying the seven valid tasks (tasks containing no bugs), six of which belong to the MinePump set. In Table 4.15 we have shown the verification times of all verifiers for two MinePump tasks, `spec1-5_product15` with an invalid verdict and `spec1-5_product62` with a valid verdict. In order to verify the tasks with a valid verdict, Jip needs to explore all execution paths. In the case of `spec1-5_product62` this requires reaching a depth of 1671[9]. From the verification times it appears that Java Ranger is the only verifier that also verifies the `spec1-5_product62` exhaustively.

[9]The other MinePump tasks require similar depths

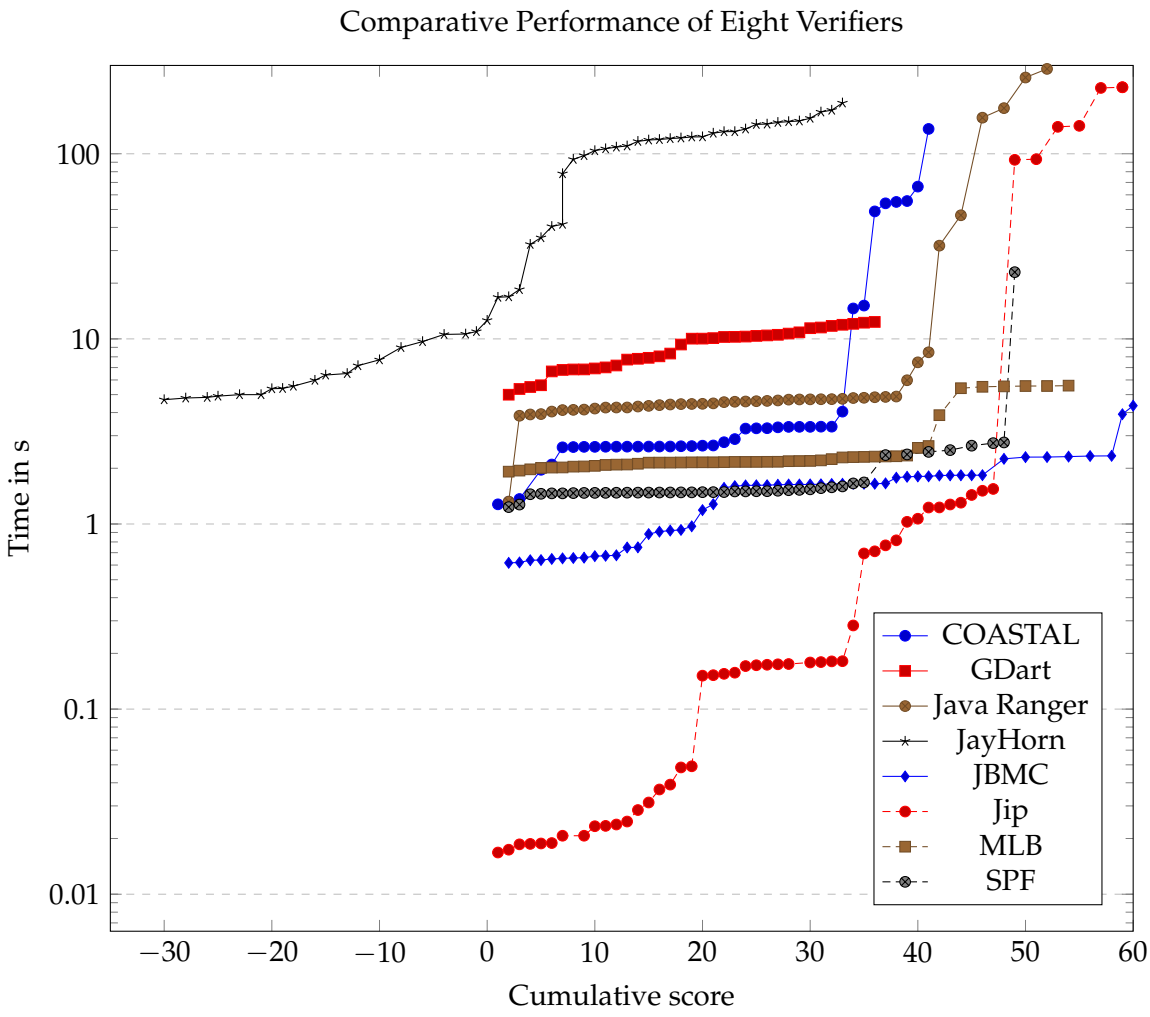

FIGURE 4.3: Quantile function (see 4.1) graph of the benchmarking results for Jip and the seven other verifiers

4.4 Soundness and Completeness

Let j denote the function mapping a task t and a verification depth d to one of three verdicts: *valid*, *invalid*, or *unknown*, executed by our SEE. Let v_{td} denote the verdict of a task t at a verification depth d. From this, we formalize soundness as:

$$j(t,d) = v_{jd} \implies v_{jd} = v_{td} \vee v_{jd} = \mathit{unknown}$$

Setting $d = \infty$ to account for programs with infinite execution paths, we define completeness as:

$$\forall \alpha \,.\, v_{t\infty} = \alpha \implies j(t,\infty) = \alpha$$

4.4.1 Soundness

Throughout the 81 verification tasks, we were unable to identify any unsound behavior. Nevertheless, we have identified two theoretical problems with soundness in Jip, both stemming from the simplified memory model implemented. We will discuss both in the subsequent paragraphs.

Index Overlap. In Jip, arrays are treated as mappings from indexes, which are represented as expressions, to values, which are also represented as expressions. The goal is to evaluate all index expressions as far as possible on each array access and array insert. This process is discussed in more detail in subsection 3.3.6.1. This method of indexing can cause unsound behavior if one index expression could represent multiple concrete places in an array.

We have outlined a minimum example of the index overlap problem in Listing 4.4. We first create a symbolic integer within the interval $\langle 1, 2 \rangle$. Next, we initialize an array[10] and assign the number five to the array element at index `a[b]`. This results in the array `a` containing the following mapping: $[0 \mapsto 0, 1 \mapsto 0, 2 \mapsto 0, b \mapsto 5,]$, where index `b` represents both index 1 and 2. However, during the while loop, `i` is a literal, and we will

only access indexes 0, 1, and 2. Consequently, the value 5 at index `b` is never accessed, which leads to an unsound invalid verdict.

```
1  // program returns an Invalid verdict
2  static void main(int b)
3  ensures(1 <= b && b <= 2)
4  {
5      int[] a := new int[3];
6      a[b] := 5;
7
8      bool containsFive := false;
9      int i := 0;
10     while (i < #a){
11         int ai := a[i];
12         containsFive := false || ai == 5;
13         i := i + 1;
14     }
15     assert containsFive;
16
17 }
```

FIGURE 4.4: The SEE does not recognize that `i == b` and thus `a[i] == 5` in one of the while loop iterations

Reference Aliasing In Jip, the heap is accessed using concrete references, taking into account the possibility that these references can be null (see subsection 3.3.5). However, Jip does not consider the possibility that references are aliases. This becomes evident in Listing 4.5, where the Symbolic Execution Engine (SEE) returns an unsound valid verdict.

[10] A newly initialized integer array has 0 assigned to all its indexes

```
1  // program returns valid verdict
2  class Main {
3     static void main(Obj a, Obj b)
4     ensures(a != null && b != null)
5     {
6        assert a != b;
7     }
8  }
9  class Obj {}
```

FIGURE 4.5: The SEE gives both `a` and `b` concrete references that could be `null`, but it does not recognize they could be equal

4.4.2 Completeness

The fact that Jip was able to verify the same set of tasks as JBMC (provided that the timeout is set high enough) substantiates the claims of completeness. Nevertheless, the Index Overlap and Reference Alias problems can still be sources of incompleteness.

Chapter 5

Conclusion

Starting this thesis we identified three primary challenges of symbolic execution: (1) memory modeling; (2) constraint solving; and (3) path explosion. We presented Jip, our Symbolic Execution Engine (SEE), with a simplified memory model and eleven heuristics designed to address the challenges of constraint solving and path explosion. To assess the SEE's performance and validate its soundness and completeness, we subjected Jip to an extensive benchmark using a set of problems from the SV-COMP.

The results from the heuristic benchmarks showed significant performance improvements. Specifically, we observed that the Expression Evaluation algorithm of Koppier, when extended with the Interval Inferer, yielded a 50 percent increase in performance. Further enhancements from pruning and formula caching heuristics, combined with the fastest (available) SMT solvers, contributed an additional 30 percent increase in performance. Another interesting result was that the parallel solving heuristic suggested by Palikareva et al. [19] seems to be infeasible to implement in any SEE.

Moreover, our comparative analysis demonstrated that our SEE could compete with state-of-the-art tools on a comprehensive set of verification tasks. The only exception being the six valid minepump programs, which some verifiers verified in as little as 5 seconds, while Jip required up to 230 seconds.

Interestingly, the benchmarks did not expose any unsound results despite the simplified (unsound) representation of references and arrays. This finding suggests that complex

memory representation may not be necessary to identify the majority of bugs in a program. In Chapter 6 we have provided a recommendation to solve this issue.

Chapter 6

Future Work

This thesis provides a range of options for future work. In the following paragraphs, we discuss what we believe to be the most interesting possibilities.

Inheritance. Although the OOX language is not currently equipped with class inheritance, it was designed to model other object-oriented languages. We recommend extending both the OOX language and the SEE to support inheritance.

Multi-threading. The OOX language was designed with multi-threading features in mind. At present, the SEE does not support these. We recommend extending the SEE to support multi-threading features. In particular, the use of partial order reduction[15][26] and compositional reasoning[21] could help to reduce the path explosion problem introduced by multithreading.

Effective Bounds. In the case of the valid minepump program, Jip was significantly outperformed by verifiers employing bounded verification i.e., not verifying all paths. Given the success of this strategy, we recommend further experimentation with verification bounds to enhance speed while maintaining strong soundness and completeness guarantees.

Detect Path Splitting. Our SEE currently does not split states in scenarios where multiple states could exist, leading to the index overlap and reference alias problems. Given the performance of our simplified approach, we suggest the implementation of a mechanism that detects when states should be split, returning an 'unknown' verdict in such cases. This modification would trade completeness for the performance boost caused by a simplified memory model.

Adaptive Pruning. This heuristic, designed to adjust the pruning probability based on the success of previous attempts, did not perform as expected. Although it significantly outperformed constant pruning in cases where no pruning was needed, it lagged behind in situations where constant pruning was more effective. This result suggests that the slower increase in pruning probability by adaptive pruning, in programs where constant pruning performed better, led to a substantial decrease in performance. For future improvements, we recommend exploring a more fine-grained approach. This could involve starting with a higher initial pruning probability and adjusting the rate at which the pruning probability is increased, making it faster than its decrease.

Bibliography

[1] Cvc4. available at https://cvc4.github.io/downloads.html.

[2] Yices. available at https://yices.csl.sri.com/.

[3] Z3. available at https://github.com/z3prover/z3.

[4] Smt-comp 2022. https://smt-comp.github.io/2022/, 2022. Accessed: June 22, 2023.

[5] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[6] D. Beyer. Reliable and reproducible competition results with benchexec and witnesses. In *Proc. TACAS*, pages 887–904.

[7] D. Beyer. Competition on software verification and witness validation: Sv-comp 2023. In *Tools and Algorithms for the Construction and Analysis of Systems: 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22–27, 2023, Proceedings, Part II*, pages 495–522. Springer, 2023.

[8] R. S. Boyer, B. Elspas, and K. N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10(6):234–245, 1975.

[9] R. Brenguier, L. Cordeiro, D. Kroening, and P. Schrammel. Jbmc: A bounded model checking tool for java bytecode. *arXiv preprint arXiv:2302.02381*, 2023.

[10] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NASA Formal Methods: 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings 7*, pages 3–11. Springer, 2015.

[11] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[12] T. Hickey, Q. Ju, and M. H. Van Emden. Interval arithmetic: From principles to implementation. *Journal of the ACM (JACM)*, 48(5):1038–1068, 2001.

[13] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf. Jayhorn: A framework for verifying java programs. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I 28*, pages 352–358. Springer, 2016.

[14] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[15] S. Koppier. The path explosion problem in symbolic execution: An approach to the effects of concurrency and aliasing. Master's thesis, Utrecht University, 2020.

[16] D. Kroening and M. Tautschnig. Cbmc–c bounded model checker: (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*, pages 389–391. Springer, 2014.

[17] X. Li, Y. Liang, H. Qian, Y.-Q. Hu, L. Bu, Y. Yu, X. Chen, and X. Li. Symbolic execution of complex program driven by machine learning based constraint solving. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 554–559, 2016.

[18] M. Mues and F. Howar. Gdart: An ensemble of tools for dynamic symbolic execution on the java virtual machine (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part II*, pages 435–439. Springer, 2022.

[19] H. Palikareva and C. Cadar. Multi-solver support in symbolic execution. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*, pages 53–68. Springer, 2013.

[20] C. S. Păsăreanu and N. Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 179–180, 2010.

[21] I. Prasetya. Formalization of variables access constraints to support compositionality of liveness properties. In *Higher Order Logic Theorem Proving and Its Applications: 6th International Workshop, HUG'93 Vancouver, BC, Canada, August 11–13, 1993 Proceedings*, pages 324–337. Springer, 1994.

[22] I. P. S. Koppier and G. Keller. O2o: an intermediate verification tool and language for concurrent object oriented programs, august 2021.

[23] V. Sharma, S. Hussein, M. W. Whalen, S. McCamant, and W. Visser. Java ranger: Statically summarizing regions for efficient symbolic execution of java. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 123–134, 2020.

[24] Tjeerd Smid. Jip. https://github.com/tjausm/Jip, 2023.

[25] W. Visser and J. Geldenhuys. Coastal: Combining concolic and fuzzing for java (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II 26*, pages 373–377. Springer, 2020.

[26] C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole partial order reduction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 382–396. Springer, 2008.

[27] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 144–154, 2012.