UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik

Bachelor's Thesis
in Computer Science

Efficient Symbolic Execution using CEGAR over Two Abstract Domains

Thomas Lemberger

Supervisor:
Prof. Dr. Dirk Beyer

July 9, 2015

Abstract

Symbolic execution is a powerful approach to automatic software verification we already applied to the domain of configurable software verification in previous work. Unfortunately, it suffers from bad runtime performance, mainly due to path explosion caused by its high precision. To mitigate this problem, we apply counterexample-guided abstraction refinement (CEGAR), an abstraction technique mostly used in model checking, to our configurable program analysis (CPA) for symbolic execution. We design two different refinement procedures for its compositional domain, considering two strongly intertwined domains at the same time. First, applying CEGAR to multiple domains is a novel approach compared to the existing single or combined refinement procedures, which only handle one domain at a time. Second, often seen as two opposites, we are, to our knowledge, the first to apply CEGAR to symbolic execution. Both refinement procedures were implemented in the verification framework CPACHECKER and evaluated with different configurations and optimizations to find the one yielding the best results. We are able to show a significant boost in runtime performance compared to symbolic execution without CEGAR for most programs. This concludes CEGAR as a valid mean to improve the runtime performance of symbolic execution and shows a valid way to apply CEGAR to multiple domains.

Contents

List of Algorithms

List of Figures

List of Tables

1 Introduction

Software systems are prone to error due to multiple factors: The developer's skills, humans' limited understanding of software principles, communication problems in development, missing or sparse documentation and unforeseen inter-dependencies between software components are just some of them.

Because of this, testing has been an integral part of software development for quite some time, often claiming about 50% of development effort and more than 50% of the budget [MSB11]. *Software testing* describes the execution of a program with the intention of finding errors. The tester, either a person or another program, uses different inputs and checks that the proper output is produced. The nature of this approach determines that only a finite amount of inputs is possible in finite time. As a result, it is impossible to ensure the error-less execution of a program with arbitrary input.

An alternative to testing is formal verification, which tries to produce formal statements that are true for all possible behaviours of a system, using mathematical methods. These statements are then used for proving that a specific specification is met. One area of formal verification is *automated software verification*. It tries to reach the above goal by only using software that works without the help or feedback of humans. CPACHECKER [BK11] is such a program that yielded excellent performance in the last iterations of the Competition on Software Verification (SV-COMP) [Bey13] [Bey14] [Bey15b]. CPACHECKER is a framework for *Configurable Software Verification* [BHT07] utilizing different *configurable program analyses* (CPAs) to locate possible property violations of a specification in a program. Three such CPAs are the value analysis CPA, which uses concrete variable assignments, the predicate CPA, which creates predicates for describing properties of program paths, and the symbolic execution CPA, which uses an extension of the value analysis CPA tracking non-deterministic values as symbolic ones in combination with the constraints CPA, which tracks constraints to symbolic values on program paths. While the value analysis CPA has high efficiency due to its simplicity, it can't handle complex program characteristics like pointers or non-deterministic values. The pred-

```c
extern __VERIFIER_nondet_int();

int main() {
  int a = __VERIFIER_nondet_int();
  int b;

  if (a >= 0) {
    b = a;

  } else {
    b = a + 1;
  }

  if (b < a) {
ERROR:
    return -1;
  }
}
```

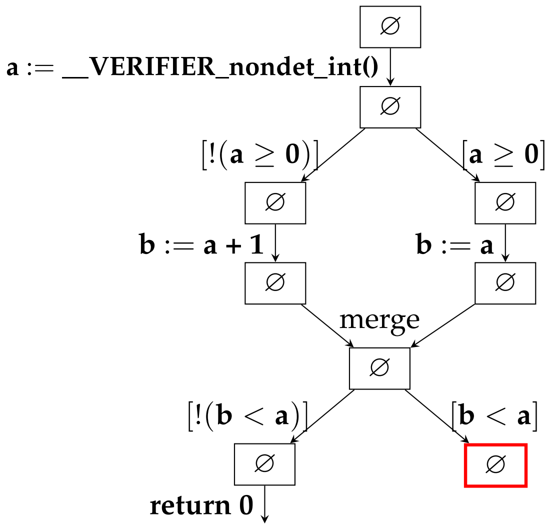


Figure 1.1: Simple program and its execution by the value analysis CPA

icate CPA, in contrast, is very expressive, but has low efficiency since satisfiability (SAT) checks are necessary for computing the feasibility of a program path. The symbolic execution CPA based on the concepts of symbolic execution [Kin76] poses something in between these two, not being able to handle some complex program characteristics as it is partly based on the value analysis CPA, but being able to handle non-deterministic values. On the other hand, it uses SAT checks, too, though less often and over smaller formulas.

Figure 1.1 displays an example program that uses non-deterministic values and its analysis using the classic value analysis CPA. Each node in the graph represents one abstract state of analysis, with the edges denoting its children. Highlighted nodes are abstract states at target locations.

Since the CPA does not store any information about non-deterministic assignments, no information about the relation between **a** and **b** exists and the property violation is reachable according to the analysis. This produces a *false alarm*. In contrast to this, the symbolic execution CPA based on symbolic execution [Kin76] tracks non-deterministic values. It can handle the assignment to **a** and its later usage.

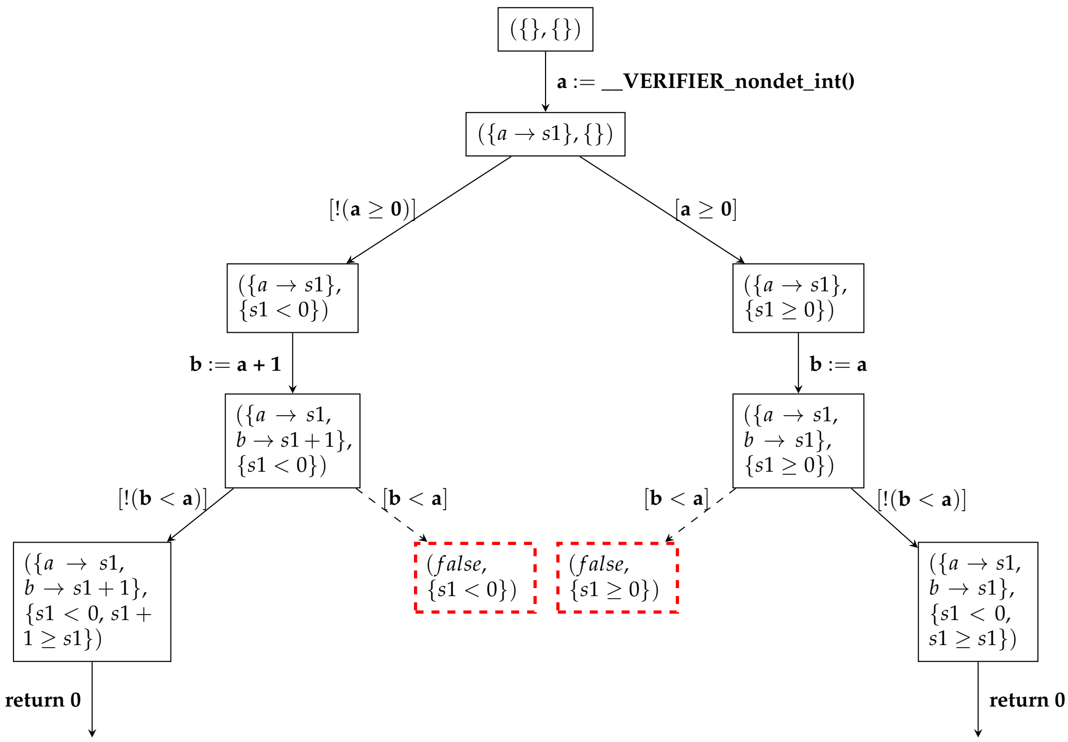


Figure 1.2: Analysis of the program in Figure 1.1 by the symbolic execution CPA

It returns that the program is safe, correctly. Figure 1.2 shows this analysis, with abstract states computed as infeasible dashed out.

Symbolic execution CPA's ability to track non-deterministic values is able to reduce the number of false alarms to a great extent, as we already showed in [Lem15]. Runtime wise, it performs poorly, though, when compared to the value analysis CPA. Since it considers all variable assignments, both deterministic and non-deterministic, and all occurring assumptions, its state space is exponential to the amount of occurring assumptions. If a infinite loop occurs, the state space is infinite, too. This problem is called *path explosion* and characteristic to symbolic execution.[AGT08] Obviously, an exponential amount of states does not scale to large programs. In addition, the cost for SAT checks, which are performed at every assumption, are exponential to the amount of non-deterministic values occurring in all encountered assumptions on the currently considered program path. Evaluation in [Lem15] showed that the symbolic execution CPA spends up to 95% of its runtime for SAT checks.

In this work we design, implement and evaluate different approaches to improving the performance of the symbolic execution CPA. Our main contribution is the application of CEGAR [CGJ+03] to the composition of the two strongly intertwined

CPAs symbolic value analysis CPA and constraints CPA (which constitute the main semantics of the symbolic execution CPA). We design two different precision refinements, both handling the two CPAs' domains and different sets of precisions successfully in one procedure. Along the way, we propose variations to the existing merge and less-or-equal operators of the constraints CPA and two different sets of precision for the same.

This work is divided into four parts: Theoretical background and contributions, their implementation, their evaluation, and future work and a conclusion. First we will describe the concepts that are the basis for our work, such as Configurable Software Verification, used CPAs and CEGAR. Next, we will illustrate the theory behind our own contribution, before explaining details about the existing and newly added implementation and deviations from theory. We will evaluate all presented concepts and compare them to the performance of the value analysis CPA, predicate CPA, and symbolic execution CPA of our old work. Last, we will give a short outlook to possible future work in this field and close with a conclusion.

2 Related Work

Symbolic execution was first introduced by [Kin76] in 1976 for program testing and verification. In this classic symbolic execution, a programming language is extended to be able to handle symbolic values without changing its syntax. Then, a program in this language is executed using symbolic values as its input. If a fork in the program control flow occurs, for example because of an if-statement, for which both branches are possible, execution splits into two parallel executions, recording the particular branching condition. Each such execution represents the execution of the program for a set of concrete input values, which can be derived based on all recorded branching conditions of an execution. This way, a lot less symbolic executions are necessary for reaching a certain test coverage than when using concrete executions. Verification at this stage was only possible by providing conditions the output of the program had to fulfill. Today, symbolic execution is used for testing, test case generation and verification.

Concolic testing In the context of testing, *concolic testing* [SMA05] [GKS05] [MS07] evolved from symbolic execution. In concolic testing, a program is executed with some arbitrary, but concrete input values, while tracking the conditions created by the branches the concrete execution takes. When finished, the last encountered, not yet negated condition is negated and new input values are created based on the conjunction of this negated condition and all other encountered conditions. The program is then executed with these new values, forced to take the previously unexplored branch the negated condition stems from. While this technique alone still suffers from path explosion, single executions are very fast as only concrete values are used, which allow easy and precise reasoning about complex data structures [BS08] and allows the simplification of constraints unsolvable using symbolic values by concrete values. In addition, the used concrete input values can be used for easy test case generation. Nevertheless, concolic testing is obviously not suitable for verification, as program properties can only be examined based on the current set of concrete input. It still deserves a mention because of its wide use and as most techniques we will present in the following use it.

There are four major areas improvements focus on for mitigating the path-explosion problem: (1) Search heuristics for achieving a high level of branch or path coverage, (2) Compositional execution, this means creating summaries of functions or paths to reuse them instead of recomputing already explored states, (3) Handling of unbounded loops, which cause infinite path exploration when using symbolic execution, and (4) Using interpolants for tracking reasons why a certain path is infeasible.

While the concepts are presented in the context of testing, they can be applied to verification, too.

Search heuristics [BS08] proposes three different heuristics for exploring a CFA in concolic testing to reach a target or uncovered branches faster, instead of simply using a depth-first search. The first heuristic chooses branches to take based on their distance in the CFA to currently uncovered branches/the target. The second heuristic, inspired by random testing[1], chooses random paths. For each branch at each execution, it is randomly decided whether to take it. The third heuristic chooses of all branches at the current path one it will not take in the next iteration, randomly. They were able to prove the increased effectiveness when using any of these three heuristics, with the first being the best in terms of coverage, quickly followed by the third. Klee [CDE08], a tool for automatic test generation running one symbolic execution for each branch taken in parallel, uses two different heuristics in turn to decide at each program location which execution to continue first. The first heuristic, called *random path selection*, maintains a binary tree representing the program path followed by all active executions. The leaves of the tree are the executions, while each node represents a fork in the program control flow. The tree is traversed from the root and each branch is taken with a possibility of 1/2. This way, executions high in the tree, which have the most freedom to reach currently uncovered branches, are more likely to be chosen. In addition, starvation is impossible due to the randomness of the heuristic. 1. Using the first heuristic increases the chance to cover previously uncovered code as soon as possible. 2. Choosing each processes at a program location with the same probability, starting at the top of the execution tree, favors executions currently high up in the tree.

[1] Testing technique using input values randomly generated

While heuristics can assist in speeding up the process of finding an error, they hardly mitigate the problem of path explosion when trying to prove that a program is error-free.

Compositional execution Compositional symbolic execution [God07] tests functions in isolation to create summaries of them. A summary of a function is a formula describing preconditions for its input and postconditions for its output. If the preconditions are met for the current used input, the function summary can be used instead of executing the function again. Summaries are computed for functions whenever no fitting summary exists, based on their call hierarchy. This use of summaries is implemented as an extension to the symbolic execution testing tool DART [GKS05], called SMART. [AGT08] extends this notion by *lazy* and *relevant exploration*, computing new function summaries only if no conjunction of summaries can be used to reach a certain branch or program location and recognizing branches that are not able to reach a certain branch or program location. It uses uninterpreted functions and predicates describing a functions (possibly not fully known) semantics for this. Thanks to its flexibility, it can be combined with any search heuristic. CPACHECKER supports block- and function summaries for the predicate CPA and value analysis CPA, so it should be easy to adapt this for the symbolic execution CPA. The use of such summaries might prove useful for programs requiring a high precision, but we assume that, when analyzing a program which only requires to track few program variables and constraint, just reducing the state space by using CEGAR and as such minimizing repeated computations is more useful. Both approaches are orthogonal though, and can be used in combination.

Handling of unbounded loops In classic symbolic execution, unbounded loops result in infinite execution. Lazy Annotation [McM10] tackles this problem by computing inductive invariants for loops, unrolling loops up to the point interpolants which are also loop invariants can prove infeasiblity of an error path. A major downside to this approach is that it will only terminate if such invariants can be found. Inspired by this approach, [JNS12] abstracts symbolic states at loop headers to only consist of invariants that hold for one path. If these invariants are too coarse to prove the infeasiblity of a counterexample, a refinement procedure similar to CEGAR is used to refine them. This is a compromise between performing eager symbolic execution and lazy CEGAR when encountering unbounded loops. [SST13] analyzes cyclic paths in the CFA and computes a so called *template* for each one, describing all possible program states that may leave the cycle after any number of iterations. Using these templates, a new *compact CFA* without any cycles is created. In the symbolic execution of this compact CFA, these templates are then used when encountering loops to directly jump to the loop exits, resulting in symbolic states based on the path to the cycle, a parameter k of iterations along the cycle, and the execution step leading to the exit. This mitigates the path explosion problem considerably, as no more loops exist in execution. In exchange, the complexity of formulas to solve deepens due to the parameter k. To prove the infeasibility of a program path based on an abstract state containing such a parameter k, the infeasibility has to be proved for all k. This introduces quantifiers to SAT checks. Evaluation shows that despite this trade-off, analysis can still be speeded up considerably.

Using CEGAR with symbolic execution mitigates the problem of unbounded loops, since no information altered by the loop is necessary, most of the time. But if it is, it results in infinite execution, also. One advantage of configurable software verification over classic symbolic execution is the possibility to combine multiple CPAs. For handling unbounded loops, a CPA specialized on doing so can be used in parallel to the symbolic execution CPA instead of extending it. A strengthening operator could be used to derive information about symbolic identifiers. In this work, we focus on the symbolic execution CPA's performance only.

Interpolation A technique closely related to the two concepts of CEGAR and the functionality of the termination check of configurable program verification is based on interpolation. If a path is found to be infeasible, an interpolant is computed for each program location on the path and stored. If such a program location is visited again on a different path, it is checked whether the interpolant is implied by the current abstract state. If it is, execution on this path can halt, as it is known that it is infeasible based on the interpolant. [JSV09] introduced this concept for the first time in the context of the Constraint Logic Programming Scheme [JMSY92]. [JNS12] stated the idea of using weakest preconditions instead of strongest postconditions for the computation of weaker interpolants in the context of verification. [JMN13] adapted it for concolic testing with arbitrary search heuristics and [CJM14] added the notion of *lazy symbolic execution*. Instead of computing interpolants immediately after a path is proven to be infeasible, execution continues on this path ignoring the infeasibility to be able to learn better interpolants. This is similar to the selection of sliced path prefixes [BLW15b] to influence the kind of interpolants that will be computed in CEGAR.

Lazy Annotation [McM10] uses interpolants to store conditions for nodes and edges on the CFA under which no target is reachable from this node or using this edge. Instead of annotating all edges on an infeasible path with interpolants in one procedure, interpolants are computed bottom-up if the current program path using this edge is infeasible up to the next possible branch.

A main difference that persists between symbolic execution with CEGAR and symbolic execution using interpolants is the amount of information stored. CEGAR is lazy, starting with a coarse precision and refining it, while symbolic execution is eager, tracking all information and computing interpolants for subsuming new states only. Using CEGAR, expensive refinements and iterative analysis happens, but the probability of a successful termination check is higher from the beginning due to the abstraction. This pays off if only few program variables/constraints have to be tracked or only few possible error paths exist by generally eliminating the path explosion problem.

CEGAR with predicate CPA/model checking and with value analysis CEGAR was introduced for boosting the speed of symbolic model checking in [CGJ+03]. First applied to the predicate CPA [BW12] in the context of configurable software verification, it was adapted to work with value analysis (e.g. the value analysis CPA) in [BL13]. While the first takes interpolants computed by an off-the-shelf SMT solver, the trial-and-error technique for computing interpolants of the latter was extended in this work for symbolic execution. Conceptually, the symbolic execution CPA is between the value analysis CPA and predicate CPA. Just like the value analysis CPA, it uses abstract variable assignments to track the explicit values of variables, where possible. In addition, symbolic values are tracked for values of unknown explicit value. The predicate CPA creates a boolean formula over a program path's statements and assumptions and checks its satisfiability. Similarly, the symbolic execution CPA tracks constraints created by assumptions on a path to derive more information about symbolic values. A SMT solver is needed to check the satisfiability of both. In contrast to the predicate CPA, formulas are not based on program variables, but on symbolic values/identifiers. Due to this, no transformation to a single static assignment form (SSA) is necessary for SAT checks. In addition, since only conditions of assumptions are tracked as constraints, boolean formulas are significantly smaller in the symbolic execution CPA and as such faster to solve. Still, the predicate CPA is a sophisticated and matured CPA that supports a lot of features and optimizations.

3 Theoretical Background

3.1 General Overview of Configurable Program Analysis

For the sake of simplicity, all theoretical concepts are based on a fictional programming language that only consists of variable assignments (e.g. $x := 5$ or $y := x$) and assumptions (e.g. $[x > 5]$ or $[y < x]$). All values are integers of arbitrary magnitude. The implementation of our presented concepts is performed in CPACHECKER, a verification tool for C programs.

We represent a program by a *control flow automaton* (CFA) [BGS][BW12]. A CFA $A = (L, l_0, G)$ is a directed graph whose nodes L represent the program locations of the program. The initial node $l_0 \in L$ represents the program entry. An edge $g \in G \subseteq L \times Ops \times L$ exists between two nodes if a program statement exists that transfers control between the program locations represented by the nodes. Each edge is labeled with the operation that transfers the control. If a node has no leaving edges, it is a *final node*. Final nodes represent the program exit. A CFA for the previously mentioned example program can be seen in Figure 3.1. A *path* σ [BLW15b] is a sequence $\langle(op_1, l_1), ..., (op_n, l_n)$ of program locations and their corresponding operations. A path σ is a *program path* if σ represents a syntactic walk through the CFA, that means for every $1 \leq i \leq n$ a CFA edge $g = (l_{i-1}, op_i, l_i)$ exists and l_0 is the initial program location. Every path $\sigma = \langle(op_1, l_1), ..., (op_n, l_n)\rangle$ defines a *constraint sequence* $\gamma_\sigma = \langle op_1, ..., op_n\rangle$. The *conjunction* of two constraint sequences $\gamma = \langle op_1, ..., op_n\rangle$ and $\gamma' = \langle op'_1, .., op'_n\rangle$ is defined as their concatenation, that means $\gamma \wedge \gamma' = \langle op_1, ..., op_n, op'_1, ..., op'_n\rangle$. The set X is the set of all program variables occurring in a program.

```
extern __VERIFIER_nondet_int();

int main() {
  int a = __VERIFIER_nondet_int();
  int b;

  if (a >= 0) {
    b = a;

  } else {
    b = a + 1;
  }

  if (b < a) {
ERROR:
    return -1;
  }
}
```

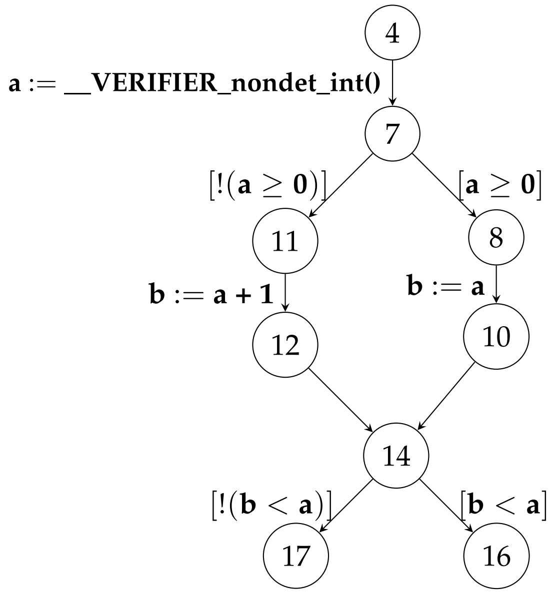


Figure 3.1: An example program and a CFA representing it

3.1.1 Concrete state

A *concrete state* c is a total function $c : X \cup \{pc\} \rightarrow \mathbb{Z}$ that assigns a specific value of $\mathbb{Z}$ to every program variable $x \in X$ and to the program counter pc. The program counter pc represents the current location in the program. The set of all concrete states of a program is C. A set $r \subseteq C$ is called a *region*. A region of concrete states that violate a given specification is called *target region* σ^t.

3.1.2 Abstract state

An *abstract domain* [BHT07] $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of a set of possible concrete states C, a semi-lattice $\mathcal{E}$ that describes the abstract states and their possible relation to each other and a concretization function $[\![\cdot]\!] : \mathcal{E} \rightarrow 2^C$ which maps each element of $\mathcal{E}$ to a subset of C.

A *semi-lattice* $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ consists of a set E of elements, a top element $\top \in E$, a partial order $\sqsubseteq \subseteq (E \times E)$ and the total function $\sqcup : (E \times E) \rightarrow E$ called *join operator*. The elements $e \in E$ of an abstract domain are called *abstract states*.

Two approaches for software verification are *model checking* and *program analysis*, also called data flow analysis. While model checking is mostly concerned with finding a program abstraction with a precision high enough to eliminate false alarms, program analysis tries to reach high efficiency by looking at only a few chosen characteristics of a program.

Program analysis starts with an initial abstract state, usually $\top$, and uses a transfer relation to derive new abstract states from old abstract states and program statements. Customization of program analysis usually means to choose one or more abstract interpreters, that is the abstract domains, transfer functions and widening operators to use.[BHT07]

Configurable software verification tries to bridge the gap of precision finding of model checking and the efficiency focus of program analysis to allow for arbitrary algorithms between these two extremes by providing the possibility to control the precision and efficiency of the algorithm by choosing all of the following:

a) one or more abstract domains to work in and the transfer functions that describe the possible transfers between abstract states,

b) a set of precision that describes the degree of abstraction within the abstract domains,

c) a *merge operator* which controls when two abstract states are merged,

d) a *stop operator* that controls when the exploration of a path is stopped, i.e. when a state is already covered by the existing reached states (this is also called termination check), and

e) a *precision adjustment operator* that can weaken or strengthen an abstract state based on a precision.

These elements are encapsulated in a configurable program analysis [BHT08], which is used by the CPA algorithm.

3.1.3 Configurable program analysis

A *CPA with dynamic precision adjustment* $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ consists of:

1. The abstract domain D, as described above. E is the set of its semi-lattice's elements. For soundness (i.e. if a property violation exists, it is always found)

and progress of the program analysis, the following requirements have to be fulfilled:[BHT07][BHT08]

a) the top element of abstract states has to represent all possible concrete states and the bottom element must represent none, formally put $[\![\top]\!] = C$ and $[\![\bot]\!] = \emptyset$,

b) the join operator has to be precise or over-approximating. That means the join of two abstract states always has to represent the same or more concrete states than the union of the concrete states both abstract states represent. This can be formally expressed as $\forall e, e' \in E : [\![e \sqcup e']\!] \supseteq [\![e]\!] \cup [\![e']\!]$, and

c) if one abstract state e is smaller than another abstract state e', the concrete states it represents must be a subset of the concrete states e' represents. Formally, that is $\forall e, e' \in E : e \sqsubseteq e' \Rightarrow [\![e]\!] \subseteq [\![e']\!]$

2. A set Π of precisions.

3. The *transfer relation* $\rightsquigarrow \subseteq (E \times G \times E \times \Pi)$. It assigns to each abstract state $e \in E$ possible abstract successors $e' \in E$ with a precision $\pi \in \Pi$. For every program statement $g \in G$ we write $e \overset{g}{\rightsquigarrow} (e', \pi)$ if $(e, g, e', \pi) \in \rightsquigarrow$ and $e \rightsquigarrow (e', \pi)$ if a program statement g with $e \overset{g}{\rightsquigarrow} (e', \pi)$ exists.

   The transfer relation $\rightsquigarrow$ has to be total, that is $\forall e \in E : \exists e' \in E : e \rightsquigarrow e'$, and it has to be precise or over-approximating. That means the union of all concrete states represented by all possible abstract successors of an abstract state e and a program statement g have to be the same or more than the union of all concrete successors of statement g and all concrete states represented by e. This can be formally expressed as $\forall e \in E, g \in G : \bigcup_{e \overset{g}{\rightsquigarrow} e'} [\![e']\!] \supseteq \bigcup_{c \in [\![e]\!]} \{c' \mid c \xrightarrow{g} c'\}$

4. The merge operator $\mathsf{merge} : E \times E \times \Pi \rightarrow E$, which weakens the information of the second given state based on the first state. It returns an abstract state of the given precision. The result of $\mathsf{merge}(e, e', \pi)$ can be anywhere between e' and $\top$. Two common merge operators are

$$\mathsf{merge}^{sep}(e, e', \pi) = e'$$

   and

$$\mathsf{merge}^{join}(e, e', \pi) = e \sqcup e'.$$

5. The stop operator $\mathsf{stop} : E \times 2^E \times \Pi \to \mathbb{B}$, which checks if the given abstract state with the given precision is covered by the set of abstract states given as second parameter. The value $\mathsf{stop}(e, R, \pi) = \mathit{true}$ always has to imply $[\![e]\!] \subseteq \bigcup_{e' \in R} [\![e']\!]$ to ensure soundness. Two common stop operators are

$$\mathsf{stop}^{sep}(e, R, \pi) = \exists e' \in R : e \sqsubseteq e'$$

and

$$\mathsf{stop}^{join}(e, R, \pi) = e \sqsubseteq \bigsqcup_{e' \in R} e'.$$

For stop^{join}, the abstract domain has to be a powerset domain, that means $e \sqsubseteq e' \Rightarrow e \supseteq e'$ for abstract states e, e'.

6. The precision adjustment $\mathsf{prec} : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$. It computes a new abstract state and precision for a given abstract state, a given precision and a given set of abstract states with precision. It can both strengthen and weaken an abstract state. The following condition has to hold:

$$\forall e, \hat{e} \in E, \pi, \hat{\pi} \in \Pi, R \subseteq (E \times \Pi) : (\hat{e}, \hat{\pi}) = \mathsf{prec}(e, \pi, R) \Rightarrow [\![e]\!] \subseteq [\![\hat{e}]\!].$$

3.1.4 CPA algorithm

The *CPA algorithm* displayed in Algorithm 1 uses an arbitrary CPA of this form to verify programs. Given a CPA $\mathbb{D}$, an initial set R_0 of reached abstract states with their precisions and a subset $W_0 \subseteq R_0$ of abstract states that have to be examined, the algorithm computes the set `reached` of abstract states reachable without encountering a target state and the currently unprocessed abstract states of that set. After initializing `reached` and `waitlist` with the given sets R_0 and W_0, the following steps are repeated until `waitlist` is empty: An abstract state $e \in$ `waitlist` is removed from the waitlist and each possible abstract successor e' and its precision π is examined:

First, precision adjustment is performed on e' based on π and `reached`. This produces a new abstract state $\hat{e}$ and a new precision $\hat{\pi}$.

Next, it is checked whether the adjusted abstract state $\hat{e}$ is a target state, i.e. represents any concrete state that violates a property. This is done by `isTargetState` $: E \to \mathbb{B}$ (Line 8,

Algorithm 1 $CPA(\mathbb{D}, R_0, W_0)$, adapted from [BL13]

Input: a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$, a set $R_0 \subseteq (E \times \Pi)$ of abstract states with their precision and a subset $W_0 \subseteq R_0$ of frontier abstract states with their precision, with E being the set of elements of D.

Output: a set of reachable abstract states with precision and a subset of it of the frontier abstract states with precision

Variables: `reached` and `waitlist`, both subsets of $E \times \Pi$

1: `reached` $:= R_0$
2: `waitlist` $:= W_0$
3: **while** `waitlist` $\neq \varnothing$ **do**
4:     choose (e, π) from `waitlist`
5:     remove (e, π) from `waitlist`
6:     **for each** e' with $e \rightsquigarrow (e', \pi)$ **do**
7:         $(\hat{e}, \hat{\pi}) := \mathsf{prec}(e', \pi,$ `reached`$)$ ▷ Precision adjustment
8:         **if** `isTargetState`$(\hat{e})$ **then**
9:             **return** $($`reached` $\cup \{(\hat{e}, \hat{\pi})\}$, `waitlist` $\cup \{(\hat{e}, \hat{\pi})\})$
10:         **for each** $(e'', \pi'') \in$ `reached` **do**
11:             $e_{new} := \mathsf{merge}(\hat{e}, e'', \hat{\pi})$ ▷ Combine with existing state
12:             **if** $e_{new} \neq e''$ **then**
13:                 `waitlist` $:= ($`waitlist` $\cup \{(e_{new}, \hat{\pi})\}) \setminus \{(e'', \pi'')\}$
14:                 `reached` $:= ($`reached` $\cup \{(e_{new}, \hat{\pi})\}) \setminus \{(e'', \pi'')\}$
15:         **if** $\neg\, \mathsf{stop}(\hat{e}, \{e \mid (e, \cdot) \in$ `reached`$\}, \hat{\pi})$ **then**
16:             `waitlist` $:=$ `waitlist` $\cup \{(\hat{e}, \hat{\pi})\}$
17:             `reached` $:=$ `reached` $\cup \{(\hat{e}, \hat{\pi})\}$
18: **return** $($`reached`$, \varnothing)$

Alg. 1), whose behaviour can be defined arbitrarily. If $\hat{e}$ is a target state, `reached` and `waitlist` are returned, both containing $\hat{e}$.

Otherwise, each already reached abstract state $e'' \in$ `reached` is individually merged with the new state $\hat{e}$ with precision $\hat{\pi}$. If the merge weakened e'', it and its precision are replaced with the weakened state and the new precision $\hat{\pi}$ in `reached` and `waitlist` (Lines 11 – 14, Alg. 1). Next, the termination check checks whether the new abstract successor $\hat{e}$ is already covered by the current reached set. If it is not, it is added to `waitlist` and `reached`. After this it the algorithm continues with the next element in `waitlist`. If `waitlist` is empty, there are no more reachable states and the reached set is returned, accompanied by the empty `waitlist`.

3.2 Basic Definition of CPAs used

The following CPAs are used and extended in this work.

3.2.1 Composite CPA

It is often useful to use multiple CPAs in parallel to combine their individual strengths, mitigate their weaknesses, and simplify CPAs by separating concerns. A composition of two CPAs [BHT08] can be expressed as

$$(\mathbb{D}_1, \mathbb{D}_2, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times).$$

It consists of:

1. Two CPAs $\mathbb{D}_1$ and $\mathbb{D}_2$. The CPAs have to share the same set of concrete states C, but can differ in any other way.
2. A composite set of precisions $\Pi_\times$.
3. A composite transfer relation $\rightsquigarrow_\times \subseteq (E_1 \times E_2) \times G \times (E_1 \times E_2) \times \Pi_\times$.
4. A composite merge operator $\mathsf{merge}_\times : (E_1 \times E_2) \times (E_1 \times E_2) \times \Pi_\times \to (E_1 \times E_2)$.
5. A composite termination check $\mathsf{stop}_\times : (E_1 \times E_2) \times 2^{E_1 \times E_2} \times \Pi_\times \to \mathbb{B}$.
6. A composite precision adjustment

$$\mathsf{prec}_\times : (E_1 \times E_2) \times \Pi_\times \times 2^{(E_1 \times E_2) \times \Pi_\times} \to (E_1 \times E_2) \times \Pi_\times.$$

The three composite operators 3. – 5. use the corresponding operators of the contained CPAs $\mathbb{D}_1$ and $\mathbb{D}_2$ as well as *strengthening operators* $\downarrow_j$ and *compare relations* $\preceq_j$ with $1 \leq j \leq 2$. They only alter lattice elements through these components.

The strengthening operator $\downarrow: E_k \times E_l \to E_k$ computes a stronger abstract state of the type E_k by using the information of an abstract state of the type E_l, with $1 \leq k, l \leq 2$ and $k \neq l$. It has to meet the requirement $\downarrow(e, e') \sqsubseteq e$. The use of strengthening operators in the transfer relation $\rightsquigarrow_\times$ allows the use of a transfer relation that is stronger than the simple combination of the transfer relations of $\mathbb{D}_1$ and $\mathbb{D}_2$.

A compare relation $\preceq \subseteq E_k \times E_l$ allows the comparison of two abstract states of different types.

The composition of CPAs can be used to construct a *composite CPA*

$$\mathcal{C} = (D_\times, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times)$$

with abstract domain $D_\times = D_1 \times D_2 = (C, \mathcal{E}_\times, [\![\cdot]\!]_\times)$ and semi-lattice $\mathcal{E}_\times = \mathcal{E}_1 \times \mathcal{E}_2 = (E_1 \times E_2, (\top_1, \top_2), (\bot_1, \bot_2), \sqsubseteq_\times, \sqcup_\times)$. The semi-lattice uses the less-or-equal operator $\sqsubseteq_\times$ defined as $(e_1, e_2) \sqsubseteq_\times (e_1', e_2')$ if $e_1 \sqsubseteq_1 e_1'$ and $e_2 \sqsubseteq_2 e_2'$ and the join operator defined as $(e_1, e_2) \sqcup_\times (e_1', e_2') = (e_1 \sqcup_1 e_1', e_2 \sqcup_2 e_2')$. The concretization function $[\![\cdot]\!]_\times$ is defined as $[\![(e_1, e_2)]\!]_\times = [\![e_1]\!]_1 \cap [\![e_2]\!]_2$.

A special merge operator in this context is

$$\mathsf{merge}^{\mathsf{agree}} : (E_1 \times E_2) \times (E_1 \times E_2) \times (\Pi_1 \times \Pi_2) \to (E_1 \times E_2).$$

It uses the merge operators of each CPA on the corresponding abstract states individually, if, after the merge, every component's state is less or equal the both given states. Otherwise it behaves like merge^{sep}, i.e. no merge is performed. We extend this definition of composite CPA to allow the composition of an arbitrary number of CPAs.

$$\mathsf{merge}^{\mathsf{agree}}(e_1, e_2, e_1', e_2', \pi_1, \pi_2) =$$
$$\begin{cases} (\mathsf{merge}_1(e_1, e_1', \pi_1), \mathsf{merge}_2(e_2, e_2', \pi_2)) & \text{if } \mathsf{merge}_1(e_1, e_1', \pi_1) \sqsubseteq e_1, e_1' \text{ and} \\ & \mathsf{merge}_2(e_2, e_2', \pi_2) \sqsubseteq e_2, e_2' \\ (e_1', e_2') & \text{otherwise} \end{cases}$$

3.2.2 Location CPA

The *location CPA* [BHT08] $\mathbb{L} = (D_\mathbb{L}, \widetilde{\Pi}, \rightsquigarrow_\mathbb{L}, \mathsf{merge}^{sep}, \mathsf{stop}^{sep}, \widetilde{\mathsf{prec}})$ is a CPA that analyzes the syntactical reachability of program locations. It does not consider any semantics and is mostly used to track the program location for other CPAs by using a composite CPA. This allows for simpler CPAs since they do not have to care about location tracking individually. The location CPA contains:

1. The abstract domain $D_\mathbb{L} = (C, \mathcal{L}, [\![\cdot]\!])$. It consists of the set C of possible concrete states, the semi-lattice $\mathcal{L}$ and the concretization function $[\![\cdot]\!]$. $\mathcal{L} = (L \cup \{\top\}, \top_\mathbb{L}, \bot, \sqsubseteq, \sqcup)$ is defined by its less-or-equal operator $\sqsubseteq$, which has the following properties: $l \sqsubseteq \top_\mathbb{L}$, $l \neq l' \Rightarrow l \not\sqsubseteq l'$ and $\bot \sqsubseteq l$ for all $l, l' \in L$ (this implies $\top_\mathbb{L} \sqcup l = \top_\mathbb{L}$ and $l \sqcup l' = \top_\mathbb{L}$ for all $l, l' \in L$ with $l \neq l'$) A semi-lattice

with these properties is also called *flat semi-lattice*. The concretization function is defined as $[\![\top_{\mathbb{L}}]\!] = C$, $[\![l]\!] = \{c \in C |\ c(pc) = l\}$ for all $l \in L$.

2. The set $\widetilde{\Pi} = \{\widetilde{\pi}\}$ of a single precision that tracks all information.

3. The transfer relation $\rightsquigarrow_{\mathbb{L}}$, which has the transfer $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} (l', \widetilde{\pi})$ if $g = (l, op, l')$ for any operation op and the transfer $\top_{\mathbb{L}} \overset{g}{\rightsquigarrow}_{\mathbb{L}} (\top_{\mathbb{L}}, \widetilde{\pi})$ for all $g \in G$.

4. The already mentioned merge operator merge^{sep}, defined as $\mathsf{merge}^{sep}(l, l', \pi) = l'$ for all $l, l' \in \mathcal{L}$.

5. The already mentioned termination check stop^{sep}, defined as $\mathsf{stop}^{sep}(l, R, \pi) = \exists l' \in R : l \sqsubseteq l'$.

6. The precision adjustment $\widetilde{\mathsf{prec}}$ that does not change anything: $\widetilde{\mathsf{prec}}(l, \pi, R) = (l, \pi)$.

3.2.3 Predicate CPA

A *predicate* is a boolean formula using linear-arithmetic expressions and equality with uninterpreted functions. The *predicate CPA* [BGS] [BHT08] uses predicate abstraction [BPR02] to compute abstract states from a formula ϕ and a set π of predicates (the precision). Two different kinds of predicate abstraction exist: The *cartesian predicate abstraction* $(\phi)^{\pi}_{C}$ is the strongest conjunction of predicates from π that is implied by ϕ. The *boolean predicate abstraction* $(\phi)^{\pi}_{B}$ is the strongest boolean combination of predicates from π that is implied by ϕ. In this work, we will only look at cartesian predicate abstraction because of its greater simplicity. For a set $r \subseteq \pi$, φ_r denotes the conjunction of all predicates in r, with $\varphi_{\{\}} = true$.

The predicate CPA $\mathbb{P} = (D_{\mathbb{P}}, \Pi_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \mathsf{merge}^{sep}, \mathsf{stop}^{sep}, \widetilde{\mathsf{prec}})$ consists of:

1. The abstract domain $D_{\mathbb{P}} = (C, \mathcal{P}, [\![\cdot]\!])$ with concrete states C, the semi-lattice $\mathcal{P}$ and the concretization function $[\![\cdot]\!]$. The semi-lattice is defined by

$$\mathcal{P} = (2^{P}, \top_{\mathbb{P}}, \bot, \sqsubseteq, \sqcup).$$

Each abstract state is a finite subset $r \in P$ of predicates, with P denoting the set of quantifier-free predicates over program variables X. An abstract state can be interpreted as the conjunction of all its predicates. $\top_{\mathbb{P}} = \varnothing$ is an abstract state without any constraints (*true*) and represents all possible concrete states. The bottom element $\perp = \{false\}$ represents no concrete state. An abstract state r is less or equal to another abstract state r', if r contains all predicates of r', formally $r \sqsubseteq r'$ if $r \supseteq r'$. The join of two abstract states r, r' is defined by $r \sqcup r' = r \cap r'$.

The concretization function $[\![\cdot]\!]$ is defined by $[\![r]\!] = \{c \in C | \ c \vDash \varphi_r\}$.

2. The precisions of set $\Pi_{\mathbb{P}} = 2^P$ describe the precision of an abstract state as a set of predicates. If predicate $p \in P$ is in a precision π, p is tracked by the analysis when π is used.

3. The transfer relation $\rightsquigarrow_{\mathbb{P}}$ has the transfer $r \overset{g}{\rightsquigarrow}_{\mathbb{P}} (r', \pi)$ if $\texttt{post}(\varphi_r, g)$ is satisfiable and r' is the largest set of predicates from π so that $\varphi_r \Rightarrow \texttt{pre}(p, g)$ for each $p \in r'$. The operations $\texttt{post}(\varphi, g)$ and $\texttt{pre}(\varphi, g)$ describe the strongest post-condition and the weakest pre-condition for a formula φ and an operation g. They are defined such that

$$[\![\texttt{post}(\varphi, g)]\!] = \{c' \in C | \ \exists c \in C : c \overset{g}{\rightsquigarrow} c' \wedge c \vDash \varphi\}$$

and

$$[\![\texttt{pre}(\varphi, g)]\!] = \{c \in C | \ \exists c' \in C : c \overset{g}{\rightsquigarrow} c' \wedge c' \vDash \varphi\}.$$

4. The already mentioned merge operator merge^{sep}, defined as $\mathsf{merge}^{sep}(r, r', \pi) = r'$ for all $r, r' \in \mathcal{P}$.

5. The already mentioned termination check stop^{sep}, defined as $\mathsf{stop}^{sep}(r, R, \pi) = \exists r' \in R : r \sqsubseteq r'$.

6. The precision adjustment $\widetilde{\mathsf{prec}}$ that does not change anything: $\widetilde{\mathsf{prec}}(r, \pi, R) = (r, \pi)$.

3.2.4 Value analysis CPA

The *value analysis CPA* [BL13] tracks integer values for all program variables with a known value explicitly. It does so by using *abstract variable assignments* [BL13]. An abstract variable assignment $v : X \nrightarrow \mathbb{Z} \cup \{\perp\}$ is a partial function that maps program variables $x \in X$ to integer values - if their assignment is known - or to $\perp$, if no possible value assignment exists. An abstract variable assignment v is *contradicting* if $v(x) = \perp$ for any $x \in \texttt{def}(v)$. For two abstract variable assignments v and v', v is *implied* by v', that is $v' \Rightarrow v$, if v' is contradicting or if $\texttt{def}(v') \subseteq \texttt{def}(v)$

and for each variable $x \in \mathtt{def}(v) \cap \mathtt{def}(v') : v(x) = v'(x)$. The *conjunction* $v \wedge v'$ is defined as

$$(v \wedge v')(x) = \begin{cases} \bot & \text{if } x \in \mathtt{def}(v) \cap \mathtt{def}(v') \text{ and } v(x) \neq v'(x) \\ v(x) & \text{if } x \in \mathtt{def}(v) \\ v'(x) & \text{if } x \in \mathtt{def}(v') \end{cases}$$

We define the *definition range* of a partial function f as $\mathtt{def}(f) = \{x \mid \exists y : (x, y) \in f\}$ and the *restriction* of a partial function f to a new definition range Y as $f_{|Y} = f \cap (Y \times (\mathcal{Z} \cup \{\bot\}))$.

The value analysis CPA $\mathbb{C} = (D_{\mathbb{C}}, \Pi_{\mathbb{C}}, \rightsquigarrow_{\mathbb{C}}, \mathsf{merge}^{sep}, \mathsf{stop}^{sep}, \widetilde{\mathsf{prec}})$ consists of the following components:

1. The abstract domain $D_{\mathbb{C}} = (C, \mathcal{V}, [\![\cdot]\!])$ contains the set C of possible concrete states, the semi-lattice $\mathcal{V}$ and the concretization function $[\![\cdot]\!]$. The semi-lattice $\mathcal{V} = (V, \top_{\mathbb{C}}, \bot, \sqsubseteq, \sqcup)$ consists of the set $V = X \nrightarrow \mathcal{Z}$ of abstract variable assignments, with X being the set of program variables and $\mathcal{Z} = \mathbb{Z} \cup \{\bot_{\mathcal{Z}}\}$ the set of integer values and the bottom element. The top element $\top_{\mathbb{C}}$ of the abstract domain is defined as $\top_{\mathbb{C}} = \varnothing$. It represents an unknown assignment for all program variables. The bottom element $\bot$ is defined as $\bot(x) = \bot_{\mathcal{Z}}$ for all $x \in X$. It represents an impossible variable assignment, that is a state that cannot be reached in the program execution. The less-or-equal operator $\sqsubseteq \subseteq V \times V$ defines $v \sqsubseteq v'$ if $\mathtt{def}(v') \subseteq \mathtt{def}(v)$ and for all $x \in \mathtt{def}(v') : v(x) = v'(x)$ or $v(x) = \bot_{\mathcal{Z}}$. This means that an abstract state v is less or equal to an abstract state v' if v contains all variable assignments of v' or restricts them even further.

   The join operator $\sqcup$ defines the least upper bound of two abstract variable assignments, but is never used in our configuration.

   The concretization function $[\![\cdot]\!]$ assigns to each abstract state v the concrete states it represents, $[\![v]\!] = \{c \in C \mid c(x) = v(x) \text{ for all } x \in \mathtt{def}(v)\}$. If an abstract state v contains an impossible variable assignment, that is $v(x) = \bot_{\mathcal{Z}}$ for any $x \in \mathtt{def}(v)$, then it represents no concrete state: $[\![v]\!] = \varnothing$.

2. The set of precisions $\Pi_{\mathbb{C}} = L \to 2^X$. A precision $\pi \in \Pi_{\mathbb{C}}$ specifies for each program location $l \in L$ a subset of program variables of X that are tracked at this location.

3. The transfer relation $\rightsquigarrow_{\mathbb{C}}$ has the transfer $v \overset{g}{\rightsquigarrow}_{\mathbb{C}} (v', \pi)$ if one of the following is true:

   a) $g = (\cdot, \texttt{assume}(p), \cdot)$ and for all $x \in \texttt{def}(v')$:

   $$v'(x) = \begin{cases} \perp_{\mathbb{Z}} & \text{if } \exists y \in X : v(y) = \perp_{\mathbb{Z}} \text{ or } p_{/v} \text{ unsatisfiable} \\ c & \text{if } c \text{ is the only satisfying assignment of } p_{/v} \text{ for } x \\ v(x) & \text{if none of the above and } x \in \texttt{def}(v) \end{cases}$$

   $p_{/v}$ is the interpretation of a predicate p using the known variable assignments of v, that is $p_{/v} = p \wedge \bigwedge_{x \in \texttt{def}(v), v(x) \in \mathbb{Z}} x = v(x) \wedge \neg \exists x \in \texttt{def}(v) : v(x) = \perp_{\mathbb{Z}}$.

   b) $g = (\cdot, w := exp, \cdot)$ and

   $$v'(x) = \begin{cases} exp_{/v} & \text{if } x = w \text{ and } exp_{/v} \neq \top_{\mathbb{Z}} \\ v(x) & \text{if } x \neq w \text{ and } x \in \texttt{def}(v) \end{cases}$$

   $exp_{/v}$ denotes the interpretation of an expression exp using the values of abstract variable assignment v, that is

   $$exp_{/v} = \begin{cases} \perp_{\mathbb{Z}} & \text{if } \exists y : v(y) = \perp_{\mathbb{Z}} \\ \top_{\mathbb{Z}} & \text{if } y \notin \texttt{def}(v) \text{ for some } y \in X \text{ that occurs in } exp \\ c & \text{otherwise, where expression } exp \text{ is evaluated to } c \text{ after replacing each occurrence of variable } x \in \texttt{def}(v) \text{ in } exp \text{ with } v(x) \end{cases}$$

   with $\top_{\mathbb{Z}}$ denoting an unknown value.

4. The merge operator merge^{sep}. That means that no merge is performed. This is the only aspect in which constant propagation CPA [BGS] and value analysis CPA differ.

5. The termination check stop^{sep}, which checks every abstract state individually.

6. The precision adjustment $\widetilde{\mathsf{prec}}$ that does not change anything: $\widetilde{\mathsf{prec}}(v, \pi, R) = (v, \pi)$. Since we only track the program location in the location CPA, a composite precision adjustment has to handle the correct adjustment of abstract states to precisions of $\Pi_{\mathbb{C}}$.

3.2.5 Symbolic value analysis CPA

The *symbolic value analysis CPA* (introduced in [Lem15] without dynamic precision adjustment and using a different less-or-equal operator) is an extension to the value analysis CPA. It introduces *symbolic values* to handle non-deterministic values and expressions of unknown value.

The symbolic value analysis CPA $\mathbb{C}_S = (D_{\mathbb{C}_S}, \Pi_{\mathbb{C}_S}, \rightsquigarrow_{\mathbb{C}_S}, \mathsf{merge}_{\mathbb{C}_S}, \mathsf{stop}^{sep}, \widetilde{\mathsf{prec}})$ consists of:

1. The abstract domain $D_{\mathbb{C}_S} = (C, \mathcal{E}, [\![\cdot]\!])$ with the set C of concrete states, the semi-lattice $\mathcal{E}$ and the concretization function $[\![\cdot]\!]$. The semi-lattice is defined as $\mathcal{E} = (V_{\mathbb{C}_S}, \top_{\mathbb{C}_S}, \bot, \sqsubseteq, \sqcup)$. Its elements are *abstract symbolic value assignments* $V_{\mathbb{C}_S} = X \nrightarrow \mathcal{Z}_{\mathbb{C}_S}$ mapping program variables in its definition range to values of $\mathcal{Z}_{\mathbb{C}_S} = \mathbb{Z} \cup S \cup \{\bot_{\mathcal{Z}}\}$. The value range of an abstract variable assignment of the type $V_{\mathbb{C}_S}$ consists of the set $\mathbb{Z}$ of concrete integer values, the set S of symbolic values and the bottom element $\bot_{\mathcal{Z}}$, which represents an impossible variable assignment. $S = S_I \cup S_E$ consists of *symbolic identifiers* S_I and *symbolic expressions* S_E. Each expression that contains at least one symbolic identifier is a symbolic expression. The definition range of an abstract variable assignment of $V_{\mathbb{C}_S}$ consists of all program variables whose value is known as either a concrete value (of $\mathbb{Z}$), a symbolic value (of S) or as invalid ($\bot_{\mathcal{Z}}$). The top element $\top_{\mathbb{C}_S} = \varnothing$ represents no known assignment.

   The less-or-equal operator is equal to the one of the value analysis CPA, but implicitly considers symbolic values: For two abstract states $v, v' \in V_{\mathbb{C}_S}$, v is less or equal to v', i.e. $v \sqsubseteq v'$, if $\mathtt{def}(v') \subseteq \mathtt{def}(v)$ and for all $x \in \mathtt{def}(v') : v(x) = v'(x)$ or $v(x) = \bot_{\mathcal{Z}}$. Note that with this operator $v' \sqsubseteq v \Rightarrow [\![v']\!] \subseteq [\![v]\!]$, but in general $[\![v']\!] \subseteq [\![v]\!] \not\Rightarrow v' \sqsubseteq v$.

   In our previous work [Lem15] we used a more complex less-or-equal operator, using the semantics of the value analysis CPA for concrete values only, and defining a new behaviour for symbolic values. We defined it as follows: $v \sqsubseteq' v'$, if all of the following conditions hold: (a) v' must only contain value assignments also present in v, that is $\mathtt{def}(v') \subseteq \mathtt{def}(v)$, (b) for every concrete or invalid assignment of v, v' must contain the same or a weaker one, that means

$$\forall x \in \mathtt{def}(v') : v(x) \in \mathbb{Z} \cup \{\bot_{\mathcal{Z}}\} \Rightarrow v'(x) = v(x) \vee v(x) = \bot_{\mathcal{Z}}$$

and (c) a bijective function $\texttt{alias} : S_I \to S_I$ exists that maps each symbolic identifier of S_I to another symbolic identifier, so that the condition

$$\forall x \in \texttt{def}(v') : v'(x) \in S \Rightarrow v(x)$$

is true for the abstract state resulting from $v'(x)$ by replacing all $i \in S_I$ occurring in $v'(x)$ with $\texttt{alias}(i)$.

This operator can result in wrong behaviour when used in conjunction with the constraints CPA, so we will not use it in this work. An example for its wrong behaviour will be presented in Section 4.2.

The join $\sqcup : V_{\mathbb{C}_S} \times V_{\mathbb{C}_S}$ is defined as

$$(v \sqcup v')(x) = \begin{cases} v(x) & \text{if } v(x) = v'(x) \\ \bot_{\mathcal{Z}} & \text{if } v(x) = \bot_{\mathcal{Z}} \text{ or } v'(x) = \bot_{\mathcal{Z}} \end{cases}$$

for all $x \in \texttt{def}(v \sqcup v')$.

2. The set $\Pi_{\mathbb{C}_S} = L \to 2^X$ of precisions. Just like $\Pi_{\mathbb{C}}$, a precision $\pi \in \Pi$ contains for each program location all program variables of X that are tracked at this location.

3. The transfer relation $\rightsquigarrow_{\mathbb{C}_S}$ contains the transfer $v \overset{g}{\rightsquigarrow}_{\mathbb{C}_S} v''$, if one of the following conditions is true:

   a) $g = (\cdot, \texttt{assume}(p), \cdot)$, $p_{/v}$ is satisfiable and for all $x \in \texttt{def}(v'')$

$$v''(x) = \begin{cases} c & \text{if } c \text{ is the only satisfying assignment for } p_{/v} \text{ and } x \notin \texttt{def}(v) \\ y & \text{if } x \notin \texttt{def}(v) \text{ and } x \text{ appears in } p.\ y \in S_I \text{ is a new symbolic values that has not been used in any other state before} \\ v(x) & \text{if none of the above and } x \in \texttt{def}(v) \end{cases}$$

   $p_{/v}$ performs an over-approximation in this case, as variables with a symbolic assignment are not considered. Reminder:

$$p_{/v} = p \wedge \bigwedge_{x \in \texttt{def}(v), v(x) \in \mathbb{Z}} x = v(x) \wedge \neg\exists x \in \texttt{def}(v) : v(x) = \bot_{\mathcal{Z}}$$

b) $g = (\cdot, w := exp, \cdot)$ and

$$v''(x) = \begin{cases} exp_{/v'} & \text{if } x = w \\ v'(x) & \text{if } x \in \texttt{def}(v) \text{ and } x \neq w \end{cases}$$

with

$$v'(x) = \begin{cases} y & \text{if } x \notin \texttt{def}(v) \text{ and } x \text{ appears in } exp.\ y \in S_I \text{ is a} \\ & \text{new symbolic identifier that has not been used in} \\ & \text{any other state before} \\ v(x) & \text{if } x \in \texttt{def}(v) \end{cases}$$

and $exp_{/v'}$ defined as before. If any symbolic value occurs in exp after replacing all occurrences $x \in X$ in exp with $v'(x)$, the expression is only partially evaluated. In this case $exp_{/v'} \in S$.

c) $v'' = \top_{\mathbb{C}_S}$.

4. The use of $\mathsf{merge}_{\mathbb{C}_S} = \mathsf{merge}^{sep}$ means that no merge of abstract states is performed.

5. The termination check stop^{sep} already mentioned considers every state independently when checking for coverage.

6. The precision adjustment $\widetilde{\mathsf{prec}}$ that does not change anything: $\widetilde{\mathsf{prec}}(v, \pi, R) = (v, \pi)$. We rely on a composite CPA to perform precision adjustment, as a location is needed.

3.2.6 Constraints CPA

The *constraints CPA* (introduced in [Lem15] without dynamic precision adjustment and with a less-or-equal operator using an `alias` function) tracks constraints (i.e. boolean formulas) on symbolic identifiers created by assume edges. For this, it relies on the values provided by the symbolic value analysis CPA to partially evaluate assume edges and create constraints out of them. The constraints CPA $\mathbb{A} = (D_{\mathbb{A}}, \Pi_{\mathbb{A}}, \rightsquigarrow_{\mathbb{A}}, \mathsf{merge}_{\mathbb{A}}, \mathsf{merge}^{sep}, \widetilde{\mathsf{prec}})$ is defined by:

1. The abstract domain $D_{\mathbb{A}} = (C, \mathcal{A}, [\![\cdot]\!])$, which consists of concrete states C, the semi-lattice $\mathcal{A}$ and the concretization function $[\![\cdot]\!]$.

The abstract states described by $\mathcal{A} = (2^{\gamma}, \top_{\mathbb{A}}, \bot, \sqsubseteq_{sub}, \sqcup)$ are subsets of the set γ of all possible boolean expressions over the possible values of the symbolic value analysis CPA, $\mathcal{Z}_{\mathbb{C}_S}$, and program variables, X. This includes concrete and symbolic values. An abstract state $a \subseteq \gamma$ can be interpreted as the conjunction of all its constraints, where each symbolic identifier $i \in S_I$ is handled as a variable. The top element $\top_{\mathbb{A}} = \varnothing$ contains no constraints (it represents *true*). The bottom element $\bot$ represents a program state that can never be reached. It represents $false$. $\sqsubseteq_{sub}$ is defined in the following way: For two given states $a, a' \subseteq \gamma$, a is less or equal a', that is $a \sqsubseteq_{sub} a'$, if a contains all constraints of a', that is $a \supset a'$.

In [Lem15] we used a less-or-equal operator using an `alias` function like the symbolic value analysis CPA did, but we will not do so in this work because it does not always work with CEGAR for the same reasons as the symbolic value analysis CPA's aliasing less-or-equal operator. The join $\sqcup$ computes the least upper bound of two abstract states, but is never used.

The concretization function $[\![\cdot]\!]$ maps an abstract state to all concrete states that satisfy its constraints:

$$[\![a]\!] = \{c \in C \mid c \vDash \varphi_a\}$$

with φ_a denoting the conjunction of all predicates in a, $\varphi_a = \bigwedge_{p \in a} p$.

2. The set $\Pi_{\mathbb{A}} = L \rightarrow 2^{\gamma^+}$ of precisions. Each precision $\pi \in \Pi_{\mathbb{A}}$ contains for each program location $l \in L$ all tracked constraints, with $\gamma^+ \subseteq \gamma$ being the set of all possible boolean expressions over $\mathcal{Z}_{\mathbb{C}_S}$. The constraints of γ^+ do not contain any program variables, but the only variables occurring in them are symbolic identifiers of S_I.

   A second option is to use the location-based precision $\Pi_L \subseteq L$, which is a subset of program locations. A precision $\pi \in \Pi_L$ tracks all constraints for every location $l \in \pi$. This type of precision is less fine-grained then $\Pi_{\mathbb{A}}$ and leads to more constraints being tracked faster.

   Example: If $p = s1 > s2 + 5$ with $s1, s2 \in S_I$ was created from an edge `assume`$(a > b)$ by using an abstract variable assignment $v = \{(a, s1), (b, s2 + 5)\}$ and $\pi(l) = \{a, b\}$, then π contains all program variables p originated from and it is tracked by π.

3. The transfer relation $\rightsquigarrow_{\mathbb{A}}$ contains the transfer $a \overset{g}{\rightsquigarrow}_{\mathbb{A}} a'$ if one of the following is true:

a) $g = (\cdot, \texttt{assume}(p), \cdot)$, $a' = a \cup p$ and a does not contain any variable $x \in X$. We just add the condition of the assume as a new constraint to the abstract state. Since a may not contain any program variables we enforce that variables must always be replaced by concrete or symbolic values through strengthening before the next `assume` edge occurs. As two `assume` edges might follow each other, we even enforce immediate strengthening.

b) $g = (\cdot, w := e, \cdot)$ and $a' = a$. The constraints CPA only cares about assume edges.

4. The merge operator $\mathsf{merge}_{\mathbb{A}} = \mathsf{merge}^{sep}$. No merge is performed when the control flow meets. We will introduce an alternative merge operator later on.

5. The termination check $\mathsf{stop}^{sep}(e, R) = \exists e' \in R : e \sqsubseteq e'$ considers every reached abstract state individually.

6. The precision adjustment $\widetilde{\mathsf{prec}}$ that does not change anything: $\widetilde{\mathsf{prec}}(r, \pi, R) = (r, \pi)$. Since we only track the program location in the location CPA, a composite precision adjustment has to handle the correct adjustment of abstract states to precisions of $\Pi_{\mathbb{A}}$.

3.2.7 Symbolic execution CPA

The *symbolic execution CPA* [Lem15] is the composition of location CPA, symbolic value analysis CPA and constraints CPA. Besides connecting the location CPA to the other CPAs so abstract states can be mapped to a program location, its most important task is the definition of a strengthening operator that creates new constraints in the constraints CPA and checks their satisfiability.

The symbolic execution CPA $\mathbb{S}$ is the composite CPA implied by the composition $(\mathbb{L}, \mathbb{C}_{\mathbb{S}}, \mathbb{A}, \Pi_{\mathbb{S}}, \rightsquigarrow_{\mathbb{S}}, \mathsf{merge}_{\mathbb{S}}, \mathsf{stop}_{\mathbb{S}}, \mathsf{prec}_{\mathbb{S}})$. It consists of:

1. The three CPAs $\mathbb{L}$, $\mathbb{C}_{\mathbb{S}}$ and $\mathbb{A}$ and their abstract domains defined above.

2. The set $\Pi_{\mathbb{S}} = \Pi_{\mathbb{C}_{\mathbb{S}}} \times \Pi_{\mathbb{A}}$ of precisions that contains the individual precisions of the symbolic value analysis CPA and constraints CPA. We do not need the precision of the location CPA because it never changes.

3. The transfer relation $\rightsquigarrow_{\mathbb{S}}$. It contains the transfer $(l, v, a) \overset{g}{\rightsquigarrow}_{\mathbb{S}} (l', v', a', \pi)$, if:

    1. $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} (l', \widetilde{\pi})$.

2. $v \overset{g}{\rightsquigarrow}_{\mathbb{C}_S} (v', \pi_{\mathbb{C}_S})$.
3. One of the following two is true:
   a) $g = (\cdot, \texttt{assume}(p), \cdot)$ and $\downarrow_{\mathbb{A},\mathbb{C}_S} (a'', v') = a'$ is defined, with the existing transfer $a \overset{g}{\rightsquigarrow}_{\mathbb{A}} (a'', \pi_{\mathbb{A}})$.
   b) $g = (\cdot, w := e, \cdot)$ and $a \overset{g}{\rightsquigarrow} (a, \pi_{\mathbb{A}})$.
4. $\pi = (\widetilde{\pi}, \pi_{\mathbb{C}_S}, \pi_{\mathbb{A}})$.

It uses the strengthening operator $\downarrow_{\mathbb{A},\mathbb{C}_S}$: $2^{\gamma} \times V_{\mathbb{C}_S} \rightarrow \gamma^{+}$ that strengthens an abstract state of the constraints CPA by using an abstract state of the symbolic value analysis CPA. $\downarrow_{\mathbb{A},\mathbb{C}_S} (a'', v) = a'$ is defined if the following conditions are true:

a) a' results from a'' by first replacing all program variables x occurring in the constraints of a'' by their abstract value assignment $v(x)$ (denoted as $a''_{/v}$) and then removing all constraints that still contain program variables:

$$a' = a''_{/v} \cap \gamma^{+}.$$

$v(x)$ can be a concrete or symbolic value as well as $\perp_{\mathbb{Z}}$. We define a constraint containing $\perp_{\mathbb{Z}}$ as $false$, though, and as such, the strengthen operator is not defined if $\perp_{\mathbb{Z}}$ is occurs.

b) $\varphi_{a'}$ is satisfiable.

4. The merge operator $\mathsf{merge}_{\mathsf{S}} = \mathsf{merge}^{\mathrm{agree}}$ uses the merge operators of each CPA on the corresponding abstract states individually, if, after the merge, every component's state is less or equal the both previous states. Otherwise no merge is performed.

5. The stop operator $\mathsf{stop}_{\mathsf{S}}$ uses the stop operators of each CPA on the corresponding abstract state and reached set individually. It only returns *true* if all of them return *true*.

6. The precision adjustment operator $\mathsf{prec}_{\mathsf{S}}$ performs precision adjustment on the abstract state of the symbolic value analysis CPA and the constraints CPA. It uses the abstract state of the location CPA in both cases to get the tracked program variables/constraints for the current location.

$$\mathsf{prec}_{\mathsf{S}}(l, v, a, \pi_{\times}, R) = (l, \mathrm{prec}_{\mathbb{C}_S}(v, l, \pi_{\mathbb{C}_S}), \mathrm{prec}_{\mathbb{A}}(a, l, \pi_{\mathbb{A}}), \pi_{\times})$$

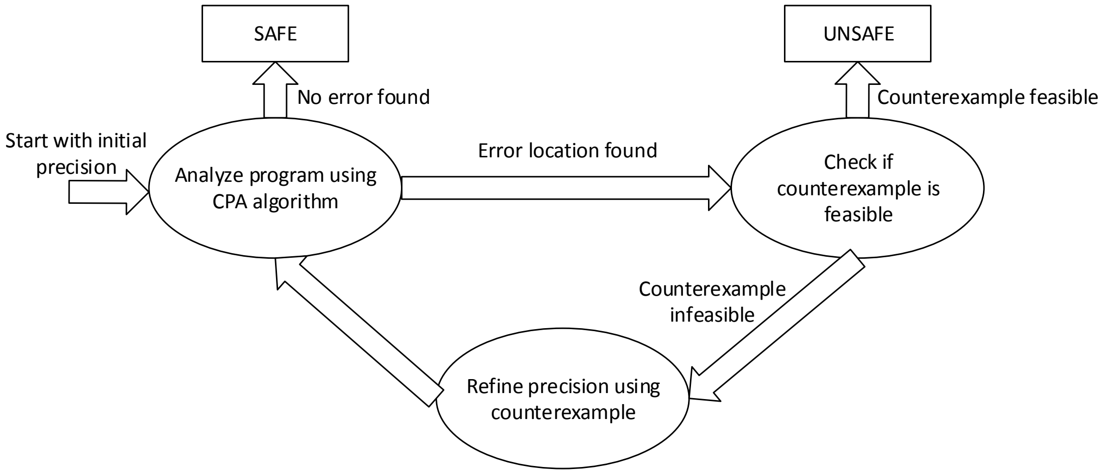


Figure 3.2: The general idea of CEGAR

with $\pi_\times = (\pi_{\mathbb{C}_S}, \pi_{\mathbb{A}})$. The precision adjustment of the symbolic value state removes the abstract variable assignments of all program variables that are not tracked by $\pi \in \Pi_{\mathbb{C}_S}$ at the current location.

$$\text{prec}_{\mathbb{C}_S}(v, l, \pi) = v_{|\pi(l)}.$$

The precision adjustment of the constraints state depends on the type of precision used for the constraints CPA: The adjustment $\text{prec}_{\mathbb{A}}$ removes all constraints that are not tracked at the current location. Its concrete implementation depends on the used set of precisions of the constraints CPA: If $\Pi_{\mathbb{A}}$ is used, which stores all tracked constraints explicitly, the adjustment is defined as

$$\text{prec}_{\mathbb{A}}(a, l, \pi) = a \cap \pi(l).$$

If $\Pi_{\mathbb{C}_S}$ is used, which only stores the program variables constraints may originate from, the adjustment deletes all constraints that originate from at least one program variable that does not occur in $\pi(l)$ with π being the current precision.

3.3 CEGAR

3.3.1 CEGAR and interpolation in general

Counterexample-guided abstraction refinement (CEGAR) [CGJ+03] is a technique to find an abstraction that contains as few information as possible while retaining the possibility to prove or disprove a program's correctness. This technique can greatly reduce the number of abstract states in a program's analysis and is considered "the most general and flexible for handling the state explosion problem,"[CGJ+03] the major problem we are facing with our symbolic execution CPA.

The technique starts analysis with a coarse abstraction and refines it based on counterexamples. A counterexample is a witness of a property violation.[BL13] If no error path is found by the analysis, it terminates and reports that no property violation exists. If an error path is found, it is checked whether the path is feasible (i.e. a possible program execution) by repeating the analysis with full precision. If the path is feasible, the analysis terminates and reports the found property violation. If the error path is infeasible it was only found because the abstraction is too coarse. As a consequence, the abstraction is refined using the error path. After this, the analysis starts again, using the new abstraction.

Since the problem of finding the coarsest possible refinement of an abstraction based on an error path is NP-hard, [CGJ+03] good heuristics have to be used to find good refinements. Interpolation [HJMM04] is one such technique originally proposed for model checking. As such stemming from a boolean context, we use interpolation for refinement of both the predicate CPA and value analysis CPA.

3.3.2 CEGAR and interpolation in the context of configurable software verification

The CEGAR algorithm displayed in Alg. 2 uses a CPA using dynamic precision adjustment $\mathbb{D}$, an initial state e_0 and an initial precision π_0 to compute whether a property violation exists.

First, the *CPA* algorithm is used to compute a set of reached abstract states (`reached`) and a subset of this set that contains all reached abstract states that have not been handled yet (`waitlist`). If `waitlist` is empty, the *CPA* algorithm has handled all reachable states without encountering any target state. If this is the case, no property violation was found and the algorithm can return $safe$. Otherwise,

Algorithm 2 $CEGAR(\mathbb{D}, e_0, \pi_0)$, adapted from [BL13]

Input: a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ with dynamic precision adjustment, an initial abstract state $e_0 \in E$ with precision $\pi_0 \in \Pi$, with E denoting the set of elements of the semi-lattice of D

Output: the verification result $safe$ or $unsafe$

Variables: the sets `reached` and `waitlist` of elements of $E \times \Pi$, an error path $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$

`reached` $:= \{(e_0, \pi_0)\}$
`waitlist` $:= \{e_0, \pi_0\}$
$\pi := \pi_0$
while true **do**
    (`reached`, `waitlist`) $:= CPA(\mathbb{D},$ `reached`, `waitlist`$)$
    if `waitlist` $= \varnothing$ **then return** $safe$
    else
        $\sigma :=$ `extractErrorPath`(`reached`)
        if `isFeasible`(σ) **then** ▷ error path feasible: report bug
            return $unsafe$
        else ▷ error path infeasible: refine and restart from the beginning
            $\pi := \pi \cup$ `refine`(σ)
            `reached` $:= (e_0, \pi)$
            `waitlist` $:= (e_0, \pi)$

an error path is extracted from the reached set. If the error path is reported as feasible, a property violation exists or the algorithm is not able to prove that none exists. It returns $unsafe$. If the error path is infeasible, the current precision is too abstract. It is refined based on the infeasible error path by using `refine` $: \Sigma \rightarrow \Pi$ with Σ being the set of all error paths, so that it can prove its infeasibility. After this, the reached set and waitlist are reset to their initial values and the algorithm repeats analysis with the refined precision. It is important to notice that the return type of `refine` has to be equal to the set Π of precisions used in $\mathbb{D}$. Because of this, CPAs are not exchangeable without changing refinement, too, in general.

For refinement, the priorly mentioned technique of interpolation is used to determine a location-specific precision that is strong enough for the CPA algorithm with precision adjustment to prove that a given error path is infeasible. A boolean formula ψ is a Craig interpolant [Cra57] for two boolean formulas γ^- (called prefix) and γ^+ (called suffix), if the following three conditions are fulfilled:

a) The prefix implies ψ, that is $\gamma^- \Rightarrow \psi$.

b) ψ contradicts the suffix, that means $\psi \wedge \gamma^+$ is contradicting.

c) ψ only contains variables occurring in *both* prefix and suffix.

It is proven that such an interpolant always exists in the domain of abstract variable assignments [BL13] as well as in the theory of linear arithmetic [Cra57].

Refinement for explicit-state model checking

Our work is strongly based on the refinement technique for abstract variable assignments. The strongest-post operator SP_{op} describes the semantics of an operation $op \in Ops$. It is the analogy to the transfer relation in the domain of CPAs. It maps a region of concrete states, implied by an abstract variable assignment, to the region of all concrete states that can be reached by executing op. The semantics of a path $\sigma = \langle (l_1, op_1), ..., (l_n, op_n) \rangle$ is defined as the consecutive application of the strongest-post operator to its constraint sequence $\gamma_\sigma = \langle op_1, ..., op_n \rangle$: $\mathsf{SP}_{\gamma_\sigma}(v) = \mathsf{SP}_{op_n}(\mathsf{SP}_{op_{n-1}}(... \ \mathsf{SP}_{op_1}(v)...))$. We use strongest-post operators during interpolation and refinement to evaluate paths.

Strongest-post Operator The strongest-post operator SP_{op} is defined in the following way: For an assignment operation $s := exp$, $\mathsf{SP}_{s:=exp}(v) = v_{|X\setminus\{s\}} \wedge v_{s:=exp}$ with $v_{s:=exp} = \{(s, exp_{/v})\}$ and $exp_{/v}$ denoting the evaluation of exp using the abstract variable assignment v, as defined in Section 3.2.4. For an assume operation $\texttt{assume}(p)$, $\mathsf{SP}_{\texttt{assume}(p)}(v) = v'$ with

$$v'(x) = \begin{cases} \bot & \text{if } \exists y \in \texttt{def}(v) : v(y) = \bot \text{ or } p_{/v} \text{ is unsatisfiable} \\ c & \text{if } c \text{ is the only satisfying assignment of } p_{/v} \text{ for } x \\ v(x) & \text{if none of the above and } x \in \texttt{def}(v) \end{cases}$$

with $p_{/v}$ as defined in Section 3.2.4.

Interpolation The algorithm for interpolation in the domain of abstract variable assignments is shown in Algorithm 3. For a prefix γ^- and a suffix γ^-, the abstract variable assignment v, that results from applying γ^- to the initial abstract variable assignment $\varnothing$ is computed. Next, for each variable assignment in v it is checked whether the assignment is necessary to prove that γ^+ is contradicting. If it is not, it can be removed from v. After all variable assignments are checked, v only contains

Algorithm 3 `interpolate`(γ^-, γ^+), adapted from [BL13]

Input: two constraint sequences γ^- and γ^+, with $\gamma^- \wedge \gamma^+$ contradicting
Output: a constraint sequence Γ, which is an interpolant for γ^- and γ^+
Variables: an abstract variable assignment v

1: $v := \mathsf{SP}_{\gamma^-}(\varnothing)$
2: **for each** $x \in \mathtt{def}(v)$ **do**
3:     **if** $\mathsf{SP}_{\gamma^+}(v_{|\mathtt{def}(v) \setminus \{x\}})$ is contradicting **then**
4:         $v := v_{|\mathtt{def}(v) \setminus \{x\}}$ ▷ x not relevant, should not occur in interpolant
5: $\Gamma := \langle \rangle$
6: **for each** $x \in \mathtt{def}(v)$ **do**
7:     $\Gamma := \Gamma \wedge \langle \mathtt{assume}(x = v(x)) \rangle$
8: **return** Γ

Algorithm 4 `refine`(σ), adapted from [BLW15b]

Input: infeasible error path $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$
Output: precision π
Variables: interpolating constraint sequence Γ

1: $\Gamma := \langle \rangle$
2: $\pi(l) := \varnothing$ for all program locations l
3: **for** $i := 1$ to $n - 1$ **do**
4:     $\gamma^+ := \langle op_{i+1}, ..., op_n \rangle$
5:     $\Gamma :=$ `interpolate`$(\Gamma \wedge \langle op_i \rangle, \gamma^+)$ ▷ inductive interpolation
6:     $\pi(l_i) :=$ `extractPrecision`(Γ)
7: **return** π

variable assignments that are necessary to prove that γ^+ is contradicting. From these, the interpolant is built (Lines 6 – 8, Alg. 3).

Refinement The interpolants produced are used in the refinement of the precision (Alg. 4). We use a location-specific precision $\pi : L \to 2^X$ that returns for a program location $l \in L$ all program variables of X which are relevant for the analysis at this location. The algorithm starts with an initial, empty interpolant Γ and empty precision π with $\pi(l) = \varnothing$ for all $l \in L$. For each location (l_i, op_i) on the error path, the suffix γ^+ of this location are set and the interpolant is computed inductively from the existing interpolant in conjunction with the current operation op_i and the suffix (Line 5, Alg. 4). A precision for the current program location is then extracted from the interpolant. One straightforward way to do this is by using all program variables with a valid assignment in the abstract variable assignment resulting from the application of the strongest-post operator to our interpolant:

$$\texttt{extractPrecision}(\Gamma) = \{x \mid (x, z) \in \mathsf{SP}_\Gamma(\varnothing) \text{ and } z \neq \bot_{\mathbb{Z}}\}.$$

It is not only sufficient, but also required to use $\Gamma \wedge \langle op_i \rangle$ instead of the full prefix $\gamma^- = \langle op_1, ..., op_1 \rangle$ for interpolation. The full prefix cannot be used as it has to be assured that the precision resulting from these consecutive interpolations proves the error path infeasible. All information necessary for proving the infeasibility of the remaining error path is present in the current interpolant and operation.

This refinement procedure can be used in CEGAR (Alg. 2) in combination with a CPA with precision adjustment that expects these precision types, like the value analysis CPA in combination with refinement for abstract variable assignments.

Lazy Abstraction Resetting the reached set and waitlist to their initial values after every refinement results in the CPA algorithm starting at the first state, always. Most of the time, this is not actually necessary though, because precision only changed for a few program locations. Because of this, lazy abstraction [HMS02] resumes analysis not at the beginning of the CFA, but at the first location that has to be revisited with its new precision so that the current error path is computed as infeasible. This location is the one before the first pair (op_i, l_i) whose corresponding interpolant is not the empty constraint sequence (i.e. the location before the first location with a new precision). We realize lazy abstraction by not resetting the waitlist and reached set in the CEGAR algorithm after each refinement procedure to (e_0, π_0). Instead, only abstract states for locations whose precision has changed and all their children are removed from the reached set and added to the waitlist. This way redundant computations without any finer precision are avoided.

Path Prefix Selection When computing the interpolant for a prefix and a suffix, the resulting interpolant is random if more than one possible interpolant exist. But some interpolants are better suited for creating a precision reaching a fast termination of analysis than others. The enhanced refinement procedure proposed by [BLW15b] allows to guide the interpolation process based on arbitrary criteria. A sliced path prefix is a path in an error path σ resulting from omitting pairs of operations and locations from the end and replacing assume operations by no-op operations. If a sliced prefix of σ is infeasible, σ is infeasible.[BLW15b]

Algorithm 5 `refine`$^{+}(\sigma)$, taken from [BLW15b]

Input: an infeasible error path $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$
Output: a precision $\pi \in L \rightarrow 2^{\Pi}$
Variables: a set Σ of infeasible sliced prefixes of σ, a mapping τ from infeasible sliced prefixes and program locations to precisions, and a sliced path prefix $\phi_{selected}$
$\Sigma :=$ `extractSlicedPrefixes`(σ) $\triangleright$ compute precisions for each infeasible sliced prefix
for each $\phi_j \in \Sigma$ **do**
  $\tau(\phi_j) :=$ `refine`(ϕ_j) $\triangleright$ Alg. 4
$\triangleright$ select suitable sliced prefix based on the prefix and its precision
$\phi_{selected} :=$ `selectSlicedPrefix`$(()\tau)$ $\triangleright$ return precision for CEGAR based on selected sliced prefix **return** $\tau(\phi_{selected})$

Algorithm 6 `extractSlicedPrefixes`(σ), taken from [BLW15b]

Input: infeasible path $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$
Output: non-empty set $\Sigma = \{\sigma_1, ..., \sigma_m\}$ of infeasible sliced prefixes of σ
Variables: a path σ_f that is always feasible
$\Sigma := \varnothing$
$\sigma_f := \langle \rangle$
for each $(op, l) \in \sigma$ **do**
  if $\mathsf{SP}_{\sigma_f \wedge (op,l)}(\varnothing) = \bot$ **then** $\triangleright$ $\sigma_f \wedge (op, l)$ is an infeasible sliced prefix
    $\Sigma := \Sigma \cup \{\sigma_f \wedge (op, l)\}$
    $\sigma_f := \sigma_f \wedge ([true], l)$ $\triangleright$ append no-op to be able to continue
  else
    $\sigma_f := \sigma_f \wedge (op, l)$ $\triangleright$ append original pair
return Σ

Algorithm 5 shows the enhanced refinement procedure. For a given error path σ, all infeasible sliced prefixes are extracted using `extractSlicedPrefixes`. For each such sliced prefix, refinement as described above is performed to derive a precision sufficient to prove σ infeasible. After this, one such precision is selected based on the prefix and its precision and returned to be used with CEGAR.

The algorithm for extracting all infeasible prefixes is shown in Alg. 6. For each operation (op, l) on a given infeasible path σ, it is checked whether it is contradicting with the already computed path σ_f, which is always feasible. If it is contradicting, the path $\sigma_f \wedge (op, l)$ is added to the set of infeasible sliced prefixes and the operation doing nothing (no-op), $[true], l)$ is appended to σ_f so that it stays feasible and new infeasible prefixes can be found using it. If it is not contradicting, σ_f is just extended by (op, l). After this, the next operand-location pair on the path is examined.

This enhanced refinement procedure allows to enhance CEGAR by selecting precisions best fit for analysis.

Refinement for the domain of linear arithmetic

Refinement in the domain of linear arithmetic, as used for the predicate CPA, uses a standard approach to refinement based on lazy abstraction and Craig interpolation. The task of interpolation is delegated to an off-the-shelf SMT solver.

In this chapter, we gave an overview of all theoretical concepts that are necessary to describe our own work. We introduced the concept of configurable software verification and configurable program analyses (CPAs), a very versatile approach to automated software verification. We introduced different CPAs we use in this work and CEGAR with precision refinement for both linear arithmetic and abstract variable assignments, which we will use when applying CEGAR to the symbolic execution CPA.

4 Efficient Symbolic Execution

To increase the performance of the symbolic execution CPA, multiple approaches are designed and evaluated. Symbolic execution suffers from two major issues: Path explosion due to its high precision and the bad performance of SAT checks. Since we use off-the-shelf SMT solvers for checking satisfiability we can not influence the performance of SAT checks. Instead almost all of our approaches focus on decreasing the state space.

We will first look at some optimizations to the existing symbolic execution CPA without using CEGAR before adapting this algorithm.

4.1 Alternative Merge Operator for Constraints CPA

For every operation $\texttt{assume}(p)$ at a location l that transfers the control flow to a location l' there exists another operation $\texttt{assume}(\neg p)$ at the same location transferring the control flow to a location $l'' \neq l'$. In most programs it is probable that the two different program branches starting at l' and l'' meet again, that means that for a later program location l''' two abstract states a, a' of the constraints CPA (in the following called *constraints states*) exist with a containing p and a' containing $\neg p$.

If a constraint p is part of an abstract state a, p is true in all concrete states represented by a (just like a predicate in an abstract state of the predicate CPA [BHT08]). If for one program location l two constraints states a, a' exist with $p \in a$ and $\neg p \in a'$ and $a \setminus \{p\} = a' \setminus \{\neg p\}$, then a represents all concrete states for which $p \wedge a \setminus \{p\}$ is true and a' represents all concrete states for which $\neg p \wedge a \setminus \{p\}$ is true. At this point, the analysis will never be able to prove a program location as infeasible because of p or $\neg p$. If a' reaches a program location and computes it as infeasible by using p, the abstract state a will compute the same program location as feasible, if it reaches it. Because of this, it seems legit to delete these obsolete constraints and only continue with one more abstract state instead of two more concrete ones by using the merge operator

$$\mathsf{merge}(a, a', \pi) = \begin{cases} a' \setminus \neg Q & \text{if } a \sqsubseteq a' \setminus \neg Q \\ a' & \text{otherwise} \end{cases}$$

with $\neg Q = \{\neg p | \ p \in a \land \neg p \in a'\}$ and $Q = \{p | \ p \in a \land \neg p \in a'\}$. It is not necessary that $a' \setminus \neg Q = a \setminus Q$. If $a' \setminus \neg Q$ represents a super set of the concrete states represented by $a \setminus Q$, that is $a \setminus Q \sqsubseteq a' \setminus \neg Q$, then the above condition is true, and $a \sqsubseteq a \setminus Q$.

This condition is automatically checked by the $\mathsf{merge}^{\mathrm{agree}}$ operator, so we can simply use $\mathsf{merge}(a, a', \pi) = a' \setminus \neg Q$. Unfortunately, we can't use this merge operator for the constraints CPA in combination with CEGAR, as our design of CEGAR does not consider merges. Only merge^{sep} is possible.

4.2 Different Less-or-equal Operators

The less-or-equal operator is the operator executed the most often during analyses as stop^{sep} uses it once for every state in the reached set, at every iteration of the CPA algorithm. In addition, it is responsible for determining whether a new state is already covered and analysis can be stopped at this point. Although the implementation framework CPACHECKER only performs a termination check for reached states at the same location, its speed and precision can make a great difference for the performance of our analysis.

Aliasing operator The less-or-equal operators we used for symbolic value analysis CPA and constraints CPA in [Lem15] using an `alias` function try to be more precise than a simple subset check. Unfortunately, they can result in false behaviour because of their independent behaviour. Consider the two pairs of value state and constraint state $e = (v, a)$ with $v = \{x \to s1, y \to s2\}$, $a = \{s1 > 0\})$ and $e' = (v', a')$ with $v' = \{x \to s2, y \to s1\}$, $a' = \{s1 > 0\})$. When using the aliasing less-or-equal operators of the symbolic value analysis CPA and of the constraints CPA, the symbolic value analysis CPA states $v \sqsubseteq v'$ for `alias` function $\texttt{alias}(s1) = s2$, $\texttt{alias}(s2) = s1$ and the constraints CPA states $a \sqsubseteq a'$ for $\texttt{alias}(s1) = s1$. Because of this, $e \sqsubseteq e'$, although the concrete states $[\![e]\!] = \{c \in C | \ c(x) > 0\}$ and $[\![e']\!] = \{c \in C | \ c(y) > 0\}$ represented by e and e' are two different sets. This violates the definition of the less-or-equal operator for abstract domains (Section 3.1.2). For this example, the less-or-equal operator of the constraints CPA actually behaves like the subset operator, since `alias` represents the identity. This shows that the less-or-equal operator of the symbolic value analysis CPA cannot be used, regardless of the operator used by the constraints CPA. Besides the default less-or-equal operator for the constraints CPA presented in Section 3.2.6, another operator might prove useful.

Implication operator Since a constraints CPA's abstract state a is interpreted as the conjunction of its constraints φ_a, it seems fit to use implication as the less-or-equal operator. Remember that $[\![a]\!] = \{c \in C |\ c \vDash \varphi_a\}$. If a formula φ_a implies a formula $\varphi_{a'}$ and c satisfies φ_a, then c also satisfies $\varphi_{a'}$. Because of this

$$[\![a]\!] = \{c \in C |\ c \vDash \varphi_a\} \subseteq \{c \in C |\ c \vDash \varphi_{a'}\} = [\![a']\!] \text{ if } \varphi_a \Rightarrow \varphi_{a'}.$$

The less-or-equal operator for the constraints CPA using implication is defined as $a \sqsubseteq_{impl} a'$ if $\varphi_a \Rightarrow \varphi_{a'}$. This operator has a higher precision than $\sqsubseteq_{sub}$ but requires SAT checks, which are definitely worse in performance than merely checking whether one set is the subset of another.

4.3 CEGAR for Symbolic Execution

For using the symbolic execution CPA with CEGAR, we have to define a refinement procedure that returns a precision of type Π_S that fits the precision of the symbolic execution CPA. We designed two such refinement procedures: The first uses adjusted versions of the refinement and interpolation algorithms as used for abstract variable assignments, with an adjusted strongest-post operator and a set of precisions that fits Π_S. The second refinement procedure extracts a precision for the symbolic execution CPA from the precision created by the refinement of the predicate CPA, which is based on interpolation in the domain of linear arithmetic.

4.3.1 Refinement based on refinement for explicit-state model checking

We adjust the refinement algorithm for abstract variable assignments (Alg. 4). The feasibility check of an error path σ is performed by executing the symbolic execution

CPA with full precision for all program locations l. If the error location of σ is reached by the analysis, the path is feasible. It is infeasible, otherwise.

We use a strongest-post operator that reflects the semantics of our symbolic execution CPA by defining a composite operator $\mathsf{SP^S}_{op} : V_{\mathbb{C}_\mathsf{S}} \times 2^{\gamma^+} \rightarrow V_{\mathbb{C}_\mathsf{S}} \times 2^{\gamma^+}$. It is the composition of the transfer relations of the symbolic value analysis CPA and the constraints CPA, as well as the strengthen operator $\downarrow_{\mathbb{A},\mathbb{C}_\mathsf{S}}$ to create useful constraints states. The result of $\mathsf{SP^S}_{op}$ is contradicting if $\downarrow_{\mathbb{A},\mathbb{C}_\mathsf{S}}$ is not defined (that means that the conjunction of constraints are contradicting) or the transfer relation of the symbolic value analysis CPA produces a contradicting abstract variable assignment. Formally:

$$\mathsf{SP^S}_{op}(v,a) = \begin{cases} (v',a'') & \text{if } v \overset{g}{\rightsquigarrow}_{\mathbb{C}_\mathsf{S}} v',\ a \overset{g}{\rightsquigarrow}_{\mathbb{A}} a',\ \downarrow_{\mathbb{A},\mathbb{C}_\mathsf{S}} (a',v') \text{ is defined with} \\ & \downarrow_{\mathbb{A},\mathbb{C}_\mathsf{S}} (a',v') = a'' \text{ and } g = (\cdot, op, \cdot) \\ \bot & \text{otherwise} \end{cases}$$

The contradiction $\bot$ represents the bottom element for both the symbolic value analysis CPA as well as for the constraints CPA. Both the transfer relation of the symbolic value analysis CPA $\rightsquigarrow_{\mathbb{C}_\mathsf{S}}$ and the transfer relation of the constraints CPA $\rightsquigarrow_{\mathbb{A}}$ always produce only one successor, so we can use them in our definition while keeping $\mathsf{SP^S}_{op}$ unambiguous. The performance of this strongest-post operator can be significantly worse than when only using abstract variable assignments since SAT checks have to be performed in the strengthen operation. Because of this, the amount of calls of the strongest-post operator can make a notable difference in performance.

Using this strongest-post operator for interpolation (Alg. 7) allows the computation of an interpolant for a prefix γ^- and a suffix γ^+ at a specific program location based on the semantics of the symbolic execution CPA. Since we want to create an interpolant Γ that contains all information necessary for proving that $\mathsf{SP^S}_{\Gamma \wedge \gamma^+}$ is contradicting, not only abstract variable assignments but also constraints have to be considered. First, we compute the strongest-post condition (v, a) for the prefix γ^- based on the initial state $\varnothing$ to get the complete prefix's strongest-post condition. Similar to interpolation for abstract variable assignments, we then eliminate all constraints from a that are not necessary for proving that γ^+ is contradicting. Next, we remove all assignments from v that are not required. This way we try to get the weakest interpolant possible. We then build the interpolant from all left constraints in a and all left assignments of v. Contrary to Algorithm 7, we build the interpolant Γ not by using `assume` operations for each $x \in \texttt{def}(v)$, but assignment operations. By using `assume` operations only for the constraints of a we can easily

Algorithm 7 $\texttt{interpolate}_{\mathbb{S}}(\gamma^-, \gamma^+)$, a modified version of Alg. 3

Input: two constraint sequences γ^- and γ^+, with $\gamma^- \wedge \gamma^+$ contradicting
Output: a constraint sequence Γ, which is an interpolant for γ^- and γ^+
Variables: an abstract variable assignment v and a constraints state a

1: $(v, a) := \mathsf{SP}^{\mathbb{S}}{}_{\gamma^-}(\varnothing)$
2: **for each** $p \in a$ **do**
3:     **if** $\mathsf{SP}^{\mathbb{S}}{}_{\gamma^+}(v, a \setminus \{p\})$ is contradicting **then**
4:         $a := a \setminus \{p\}$ ▷ p not relevant, should not occur in interpolant
5: **for each** $x \in \texttt{def}(v)$ **do**
6:     **if** $\mathsf{SP}^{\mathbb{S}}{}_{\gamma^+}(v_{|\texttt{def}(v)\setminus\{x\}}, a)$ is contradicting **then**
7:         $v := v_{|\texttt{def}(v)\setminus\{x\}}$ ▷ x not relevant, should not occur in interpolant
8: $\Gamma := \langle\rangle$
9: **for each** $p \in a$ **do**
10:     $\Gamma := \Gamma \wedge \langle \texttt{assume}(p) \rangle$
11: **for each** $x \in \texttt{def}(v)$ **do**
12:     $\Gamma := \Gamma \wedge \langle x := v(x) \rangle$
13: **return** Γ

distinguish between both domains. It is not wrong to only use `assume` operations, but it would unnecessarily increase the precision for the constraints CPA and the constraints sets used in the inductive interpolations, as all assumptions are added here. This could significantly decrease performance during interpolation due to the for-loop over all constraints in Line 9 and the strongest-post operators bad performance. It might even be more effective to just use all constraints and not perform these additional strongest-post computations by omitting Lines 2 – 4 of the algorithm. Since constraints made of assume edges whose occurring program variables have an unknown value are discarded in strengthening of the constraints CPA, the precision of the value analysis CPA indirectly influences the constraints the constraints CPA tracks. Additionally, it is also possible to eliminate variable assignments first, and constraints second. We will examine all three possibilities later in detail (Section 7.2.2).

After interpolation is done, a precision of type $\Pi_{\mathbb{S}}$ must be extracted from this interpolant so that future executions of the CPA algorithm with the symbolic execution CPA can prove the examined error path as infeasible. To get a precision that consists of the two individual precisions of symbolic value analysis CPA and constraints CPA from an interpolant Γ, we define the `extractPrecision` function

Algorithm 8 `refine`$_{\mathbb{S}}(\sigma)$, a modified version of Alg. 4

Input: infeasible error path $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$
Output: precision $(\pi_{\mathbb{C}_\mathbb{S}}, \pi_{\mathbb{A}}) \in \Pi_\mathbb{S}$
Variables: interpolating constraint sequence Γ

$\Gamma := \langle\rangle$
$(\pi_{\mathbb{C}_\mathbb{S}}(l), \pi_{\mathbb{A}}(l)) := (\varnothing, \varnothing)$ for all program locations l
for $i := 1$ to $n - 1$ **do**
  $\gamma^+ := \langle op_{i+1}, ..., op_n \rangle$
  $\Gamma :=$ `interpolate`$_\mathbb{S}(\Gamma \wedge \langle op_i \rangle, \gamma^+)$ ▷ inductive interpolation
  $(\pi_{\mathbb{C}_\mathbb{S}}(l_i), \pi_{\mathbb{A}}(l_i)) :=$ `extractPrecision`$_\mathbb{S}(\Gamma)$
return $(\pi_{\mathbb{C}_\mathbb{S}}, \pi_{\mathbb{A}})$

as the composition of two new functions, each of which extracts the precision for one of these CPAs based on Γ:

$$\texttt{extractPrecision}_\mathbb{S}(\Gamma) = (\texttt{extractPrecision}_{\mathbb{C}_\mathbb{S}}(\Gamma), \texttt{extractPrecision}_{\mathbb{A}}(\Gamma))$$

with

$$\texttt{extractPrecision}_{\mathbb{C}_\mathbb{S}}(\Gamma) = \{x \mid (x, z) \in v, z \neq \bot_\mathcal{Z} \text{ and } \mathsf{SP}^\mathbb{S}{}_\Gamma(\varnothing) = (v, a)\}$$

and

$$\texttt{extractPrecision}_{\mathbb{A}}(\Gamma) = a \text{ with } \mathsf{SP}^\mathbb{S}{}_\Gamma(\varnothing) = (v, a)$$

if the constraints CPA uses the precision $L \to 2^{\gamma^+}$, or

$$\texttt{extractPrecision}_{\mathbb{A}}(\Gamma) = \{x \mid \exists p \in a : \text{p originates from an expression with } x\}$$

if the constraints CPA uses the precision $L \to 2^X$. The symbolic value analysis CPA is based on the value analysis CPA, so it also works with the default refinement procedure for abstract variable assignments described in Section 3.3.2. We simply use the existing `extractPrecision` for this CPA. The precision of the constraints CPA is the set of all tracked constraints, so the constraints that result from applying the strongest-post operator to the interpolant provide the precision needed at the current location for proving the current error path as infeasible in future analysis. The adjusted refinement procedure is shown in Algorithm 8.

Since sliced prefix selection (Sec. 3.3.2) only relies on a strongest-post operator and an existing refinement procedure, we can easily use it with our refinement procedure and SP^S. Evaluation will show that prefix selection can boost the performance of symbolic execution with CEGAR significantly.

4.3.2 Refinement based on refinement for predicate CPA

Another possible way of refinement is to delegate the procedure to the refinement of the predicate CPA and extract a precision of type Π_S from the created predicate precision. This might not always work as the predicate CPA is able to handle more complex operations than the symbolic execution CPA, for example non-deterministic arrays. Even if an error path σ is infeasible by using the symbolic execution CPA with full precision, the interpolant (and resulting precision) computed by the predicate CPA's refinement could rely on unsupported operations.

Two other drawbacks exist due to the predicate CPA's set of precisions: In the current implementation of CPACHECKER, we are not able to create constraints out of the predicates of the precision created by the predicate CPA's refinement, but can only extract program variables' names. Due to this it is not possible to use the constraint-specific precision set $\Pi_{\mathbb{A}} = L \rightarrow 2^{\gamma^+}$ as part of Π_S, but only $\Pi_{\mathbb{C}_S}$ (see Sec. 3.2.6). Secondly, the set $\Pi_{\mathbb{P}} = 2^P$ of precisions of the predicate CPA is not location-based per definition. Because of this, we have to assign the same set of tracked program variables for each location. This is not a problem in the implementation though, because a more detailed adjustment of the predicate CPA is possible there.

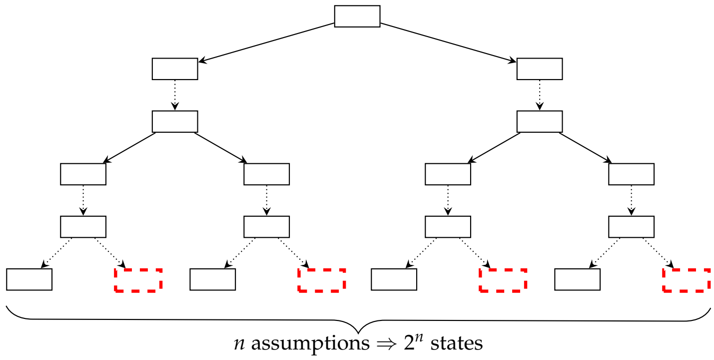
Despite these problems this approach might still yield better performance than $\texttt{refine}_S$. Though both have to rely on SMT solvers, our own procedure performs one SAT check for every constraint, while the predicate CPA's refinement utilizes a SMT solver to handle the complete interpolation process, which is presumably more performant due to its specialization. Algorithm 9 shows the refinement procedure delegating to predicate CPA's refinement. After computing the precision π' of the predicate CPA, the variables used in the predicates of π' are extracted and assigned as the precision $\pi_{\mathbb{C}_S}(l)$ for every location l. The precision $\pi_{\mathbb{C}_S}$ is then returned as precision for both symbolic value analysis CPA and constraints CPA.

Figure 4.1 shows the benefit of using CEGAR with the symbolic execution CPA. The figure shows how analysis without CEGAR creates a lot of abstract states with information not necessary for proving that, increasing exponentially with the amount of assumptions in the program. In contrast, analysis with CEGAR consists of two iterations: While the target state is reachable in the analysis with empty precision, it

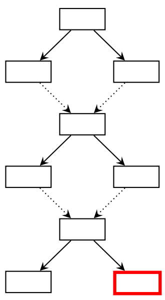
Algorithm 9 $\texttt{refine}'_{\mathbb{S}}(\sigma)$

Input: infeasible error path $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$
Output: precision $(\pi_{\mathbb{C}_\mathbb{S}}, \pi_{\mathbb{A}}) \in \Pi_{\mathbb{S}} = \Pi_{\mathbb{C}_\mathbb{S}} \times \Pi_{\mathbb{C}_\mathbb{S}}$
Variables: predicate precision $\pi' \in \Pi_{\mathbb{P}} = 2^P$

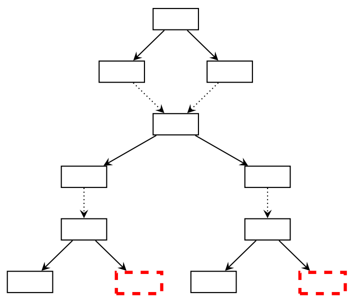
$(\pi_{\mathbb{C}_\mathbb{S}}(l), \pi_{\mathbb{A}}(l)) := (\varnothing, \varnothing)$ for all program locations l
$\pi' = \texttt{refine}_{\mathbb{P}}(\gamma)$
$\pi_{\mathbb{C}_\mathbb{S}}(l) := \texttt{extractPrecision}'_{\mathbb{S}}(\pi')$ for all l
return $(\pi_{\mathbb{C}_\mathbb{S}}, \pi_{\mathbb{C}_\mathbb{S}})$



(a) Analysis without CEGAR



(b) Analysis with CEGAR, first iteration: Empty precision



(c) Analysis with CEGAR, second iteration: Refined precision

Figure 4.1: Symbolic execution analysis of a program with and without CEGAR

is infeasible in the analysis with refined precision. Both runs themselves need far lesser abstract states than analysis without CEGAR, as only necessary information is tracked.

5 Implementation of used CPAs in CPAchecker

After defining the theoretical background of our work, deviations of the implementation from the theory and optimizations are documented next.

5.1 Basic implementation

We implemented our algorithm in the framework for configurable program verification CPACHECKER[BK11]. CPACHECKER is a command-line tool that is able to handle C programs without recursive function calls or multi-threading. It parses a C program, creates a CFA representing the program, and executes the CPA or CEGAR algorithm on it. The CPAs and, in the case of using CEGAR, the refinement procedure to use have to be defined in a configuration file that can be specified on the command line using the parameter `-config <FILE>`. The specification to check must be defined as a temporal logic formula and is represented by an own CPA. This CPA represents the `isTargetState` function of the CPA algorithm.

A wide array of CPA implementations already exists, which we can utilize. The CPA interface equals the theoretical definition, but is extended with an initial state and initial precision, which are used as initial parameters of the CPA algorithm. The interface of the transfer relation is extended to include strengthening for the designated CPA, as almost always a composite CPA is used. Instead of defining one strengthen operator for each combination of two states, as done in theory ,the strengthen method gets all states of the current composite state, so that it can choose which to use for strengthening its own state.

One implementation for the composite CPA exists, called CompositeCPA. It allows arbitrary composition of CPAs by using their transfer relations, strengthen operators, the merge-agree operator $\mathsf{merge}^{\mathrm{agree}}$, a stop operator that only returns *true* if all subordinate stop operators return *true*, and by delegating the precision adjustment to each individual CPA's precision adjustment operator not only with the precision to adjust to, but also *all* abstract states of the composite state, not only the one of the CPA delegated to. We used this CPA to create our symbolic execution CPA.

The way the precision adjustment of CompositeCPA works, it is possible to implement a precision adjustment function directly for a CPA whose precision uses location-specific tracking, like $\Pi_{\mathbb{C}_S}$. As such we implemented the two auxiliary precision functions $\text{prec}_{\mathbb{C}_S}$ and $\text{prec}_{\mathbb{A}}$ as the precision adjustment of the symbolic value analysis CPA and constraints CPA. By just specifying the wanted CPAs as components of the composite CPA in a configuration file we composed the symbolic execution CPA. We use the existing location CPA without any modifications. It was already implemented in CPACHECKER.

Both the symbolic value analysis CPA, a direct extension of the existing value analysis CPA, and the constraints CPA, a completely new CPA, were mostly used as they were implemented in our work for [Lem15]. The constraints CPA transfer relation's complete syntax is in the strengthening by the symbolic value analysis CPA to forgo the need for constraints made of program variables which are then instantly replaced with constraints made of symbolic values. This way, a constraints state always only contains constraints over explicit and symbolic values. This means that all constraints are of γ^+.

SAT checks over the constraints of a constraints state are performed by creating a conjunction of boolean formulas, each of which represents one constraint of the state, with symbolic identifiers being transformed to variables in the formulas. This conjunction is then given to a SMT solver.

To be able to handle conditions like "each constraint that originates from program variable x" (for example as used when using the set $\Pi_{\mathbb{C}_S}$ of precisions for the constraints CPA, see Section 3.2.6 and 4.3.1), we also store for each symbolic value that is assigned to an variable the variable *in* the symbolic value. This way it is always possible to transform a constraint of γ^+ back to its original representation.

5.2 Existing options/optimizations

To use symbolic values in the value analysis CPA, the configuration option

```
cpa.value.symbolic.useSymbolicValues = true
```

has to be set. Since structures and arrays in C may have a lot of entries, which might not even be important for the analysis, it can prove useful not to track them at all, if their assignments are non-deterministic. This increases the probability of coverage of a state, for example

```
someStruct a;
int b = 1;
if (__nondet_int()) {
      a = __VERIFIER_nondet_pointer();
} else {
      a = __VERIFIER_nondet_pointer();
}
if (b < 1) {
ERROR:
      return -1;
}
```

results in two different abstract states when the control flow meets and $b < 1$ is checked twice, if non-deterministic structure assignments are tracked. If they are not tracked, analysis will terminate for one abstract state when the control flow meets and unnecessary computation is avoided. The options `cpa.value.symbolic.handleArrays` as well as `cpa.value.symbolic.handleStructs` can be used to disable the tracking of non-deterministic assignments to structs and arrays. This will only disable the tracking of non-deterministic assignments of the type struct or arrays, i.e. assignments to members of structs and array elements are always tracked, despite the value of those two options. When analyzing

```
int[] b = new int[5];
b[0] = __VERIFIER_nondet_int();
```

with the symbolic execution CPA, b is always tracked. Arrays of unknown length are never tracked, though.

Multiple optimizations are applied to the constraints CPA implementation in CPAchecker. First, we use the SMT solver to create a model (a mapping of variables to concrete values) that satisfies the conjunction F of all constraints of a state. This model is used to compute all *definite assignments*, i.e. the variables for which only one valid assignment exists.

For each variable assignment $s = n$ of the model, we check whether n is the only assignment for variable s that satisfies F by checking whether $F \wedge s \neq n$ is unsatisfiable. If it is, $s = n$ is necessary for the formula to be true and we can store it as a definite assignment. (Alg. 10)

Algorithm 10 GetDefiniteAssignments(F, M)

Input: A boolean formula F and a model $M = S_I \rightarrow \mathbb{Z}$ that satisfies F
Output: A map $D \subseteq M$ of definite assignments

```
for each (s, n) ∈ M do
    if F ∧ s ≠ n is unsatisfiable then
        D := D ∪ (s, n)
return D
```

In addition, we only store constraints that contain at least one symbolic identifier. If a constraint does not contain a symbolic identifier, we call it *trivial*. Its representing boolean formula then does not contain any variables and it can be checked whether it is satisfiable or not independently of all other constraints, as it can't influence any symbolic identifier's possible concrete values. If the constraint is unsatisfiable, the path using this assumption is infeasible and no valid transfer to a new state exists. Otherwise, the old state is used without adding the trivial constraint. In our basic implementation, if a constraint that already is in the current abstract state becomes trivial because all symbolic identifiers occurring in it have definite assignments, the constraint was removed, too, while the definite assignments were preserved. This was done for the same reason as not adding trivial constraints in the first place, but resulted in more complex code, as the definite assignments had to be considered every time a constraints state was examined. For simplicity, this feature was removed from our current implementation. CEGAR automatically results in a constraints state that only contains constraints still necessary, so that such trivial constraints are removed automatically by not being tracked.

Last, we do not create boolean formulas for each constraint every time we want to perform a SAT check, but store them and only create formulas for constraints for which none exist yet. This way we have to synchronize the set of constraints with an additional set of formulas, but save a lot of redundant formula creations.

We perform strengthening of the value analysis CPA by using constraints states. If an abstract assignment of a symbolic identifier with a definite assignment to a program variable exists in the value analysis state, the symbolic identifier is replaced by the definite assignment's value. This way we reduce the number of existing symbolic identifiers to the necessary minimum and create constraints with fewer symbolic identifiers.

5.3 New options/optimizations

We extended strengthening of the value analysis CPA by the constraints CPA to simplify symbolic expressions, as long as they are independent. If none of the symbolic identifiers occurring in a symbolic expression are part of any constraint of the constraints state and no other program variable's assignment, its value can be any number, independently of all other variable assignments. In conclusion, any such expression can be replaced with a single symbolic identifier without losing any information. Such independence can be checked easily by traversing through all constraints' operands. By replacing potentially complex expressions by a simple single symbolic identifier, the occurrence of complex formulas that have to be solved in SAT checks are reduced to single variables.

By using configuration option `cpa.constraints.mergeType = SEP` or `JOIN`, either merge^{sep}, as used in [Lem15], or the new merge operator merge as defined in Section 4.1 can be used in the constraints CPA.

For choosing the less-or-equal operator to use with the constraints CPA, the property `cpa.constraints.lessOrEqualType` with possible values `SUBSET`, `ALIASED_SUBSET` and `IMPLICATION` exists. Each less-or-equal operator behaves as described in Section 4.2. Keep in mind that `ALIASED_SUBSET` can result in wrong behaviour.

6 Implementation of CEGAR

For applying CEGAR to the symbolic execution CPA, the CEGAR algorithm implementation already present in CPACHECKER is used. To use it, the configuration option `analysis.algorithm.CEGAR = true` has to be set. In addition, the refinement procedure has to be set with property `cegar.refiner`. Its value has to be the name of a class containing a method

```
public static Refiner create(ConfigurableProgramAnalysis)
```

which is called before starting the CEGAR algorithm to get the refinement procedure. The class name has to be given with its package description starting at `org.sosy_lab.cpachecker`. If the class to use were `org.sosy_lab.cpachecker.cpa.value.refiner.ValueAnalysisRefiner`, the configuration option `cegar.refiner = cpa.value.refiner.ValueAnalysisRefiner` would have to be set.

In CPACHECKER, refinement for multiple CPA's precisions is already implemented. Since our precision refinement for the symbolic execution CPA based on abstract variable assignments (Section 4.3.1) is very similar to the refinement of the value analysis CPA, we refactored it to be able to reuse most code.

6.1 Refactoring of ValueAnalysis CEGAR into Generic Form

The refinement procedures for value analysis CPA and symbolic execution CPA differ in the following ways, in theory:

1. Feasibility check `isFeasible` for error paths. The feasibility check of the value analysis refinement uses the value analysis CPA with full precision, the check of the symbolic execution refinement uses the symbolic execution CPA with full precision.

2. Set of precisions. Symbolic execution CPA's precision is a pair of the symbolic value analysis CPA's precision, which is the same as the value analysis CPA's one, and the constraints CPA's precision. This changes the expected return type of the `extractPrecision` function and the refine algorithm. Since the CEGAR implementation in CPACHECKER expects the refinement procedure to also update the precision in the CPAs, a different `extractPrecision` function and a different precision update procedure are needed.

3. Interpolation algorithm with strongest-post operator and structure of the produced interpolant. Symbolic execution uses an interpolation algorithm with another behaviour and a different structure of the returned interpolant. This has to be considered in the `extractPrecision`, also.

Keeping these points in mind, we first take a look at the old structure of the value analysis CPA's refinement implementation.

6.1.1 Structure of value analysis CPA refinement

Figure 6.1 shows the structure of the default refinement procedure for the value analysis CPA. The class `ValueAnalysisRefiner` is acting as interface for the refinement procedure. A deviation from the CEGAR algorithm (Alg. 2) is that the refinement procedure does not get the error path extracted from the reached set, but the reached set itself. The refinement procedure is responsible for extracting the error path, checking whether it is feasible, updating the precision if it is not and resetting the reached set and waitlist. (Lines 9 – 15 in Alg. 2).

Independent of the way the reached set and waitlist are reset, the precision is updated by getting the existing precisions of all locations removed from the reached set and joining them with the newly extracted precision.

A deviation from the `refine` algorithm (Alg. 4) is that the interpolants for all program locations on the error path are created by the `ValueAnalysisPath-Interpolator` in one go, stored as a `ValueAnalysisInterpolationTree`. It basically represents Lines 3 – 5 of the refine algorithm, while `ValueAnalysis-EdgeInterpolator` is used for concrete interpolation. The interpolation algorithm (Alg. 3) gets a prefix and a suffix as parameters, with the prefix being combined from the interpolant computed for the last location and the current location's operation. It then applies the strongest-post operator to the complete prefix with an initial abstract variable assignment to get the abstract variable assignment for the current

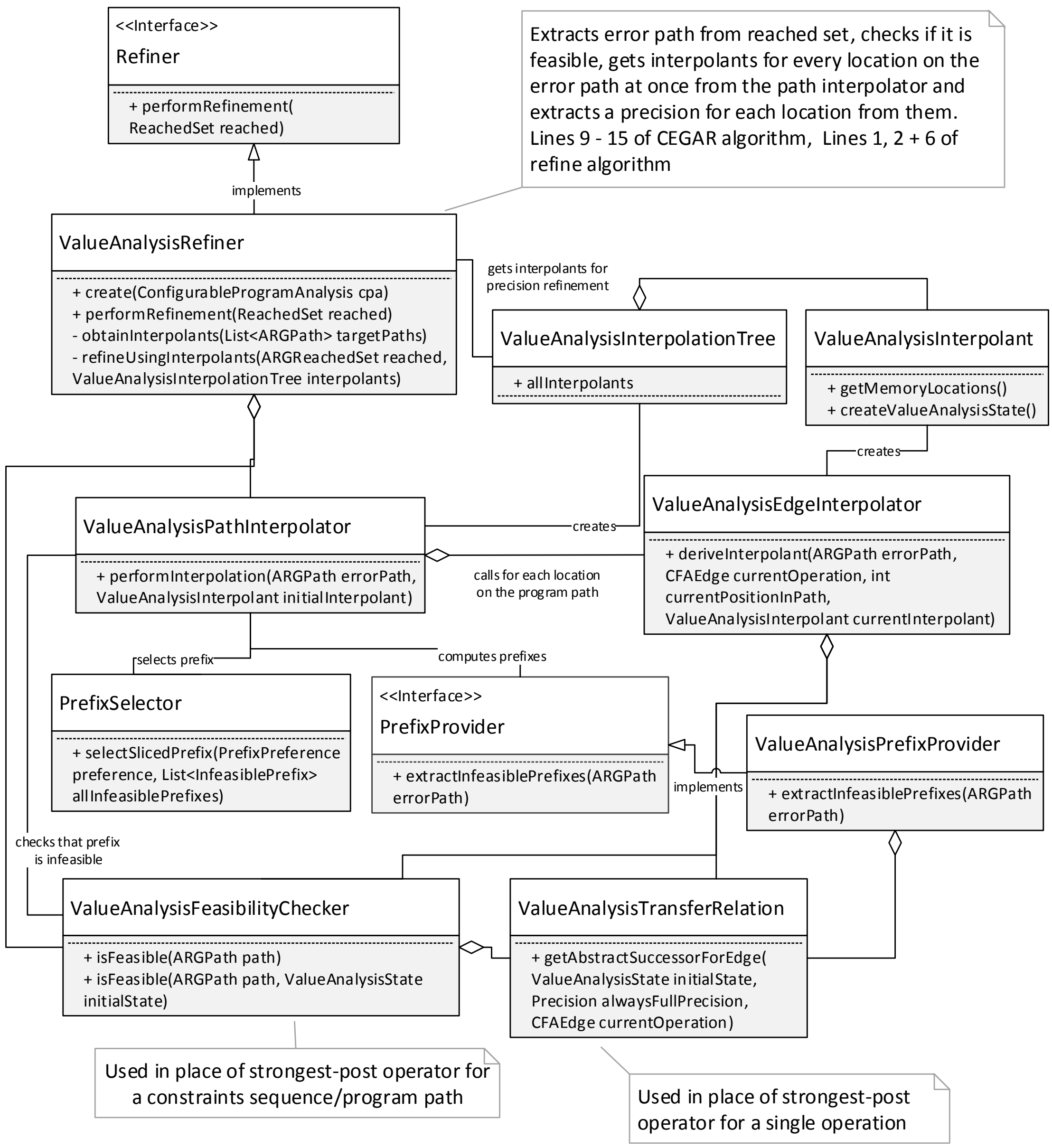


Figure 6.1: Structure of value analysis CPA refinement before refactoring

location. In the implementation, `ValueAnalysisEdgeInterpolator` receives the interpolant and the current operation in form of a CFA edge, separately. It is possible to recreate a `ValueAnalysisState` from the interpolant class `Value-AnalysisInterpolant`, so the strongest-post operator only has to be applied to the current operation with the reconstructed state as initial one. This way, no redundant computations happen.

`ValueAnalysisEdgeInterpolator` uses the `ValueAnalysisTransferRelation` with full precision as strongest-post operator SP for single operations, as it represents the same semantics. The `ValueAnalysisState` also used in the abstract domain of the value analysis CPA is used to represent abstract variable assignments. A program variable is represented as a `MemoryLocation`. The class `Value-AnalysisFeasibilityChecker` is used for the feasibility check `isFeasible` (Line 10. Alg. 2). Since it applies the `ValueAnalysisTransferRelation` also representing SP sequentially to a program path to check whether it is feasible, it is also used as the sequential application of the strongest-post operator on a program path by the `ValueAnalysisEdgeInterpolator`. It is not necessary to transform program paths to constraints sequences, as the transfer relation can work on their edges directly.

The interface `PrefixProvider`, its implementation `ValueAnalysisPrefix-Provider` and the class `PrefixSelector` are used for the selection of an infeasible path prefix of the error path. Interpolation is then applied to this prefix only instead of the whole path, to have better control of the interpolants produced and the interpolation process itself. This concept was introduced in [BLW15b] and will be used by us without modification. The algorithm for determining infeasible prefixes also uses the strongest-post operator. In the implementation, `ValueAnalysis-PrefixProvider` uses the `ValueAnalysisTransferRelation` for this, as all others do. In addition, the infeasibility of a chosen prefix is checked again by using the `ValueAnalysisFeasibilityChecker` since it is possible to be feasible due to imprecision when structs or arrays occur.

6.1.2 Introduction of interfaces

Except for the two classes `ValueAnalysisRefiner` and `ValueAnalysisPrefix-Provider`, none of these classes are accessed through an interface. So first, we created interfaces with type parameters that represent all components required for refinement in the domain of abstract variable assignments, based on the existing imple-

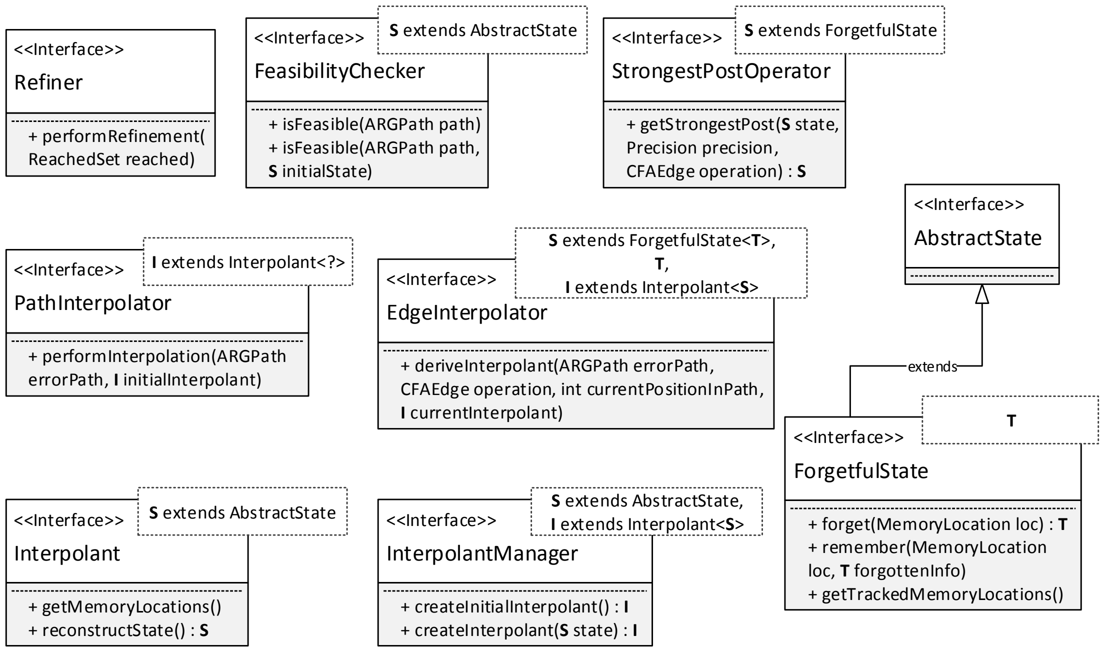


Figure 6.2: Interfaces used in refinement

mentation of the value analysis refinement. These interfaces are displayed in Figure 6.2 next to the `Refiner` interface. `FeasibilityChecker`, `PathInterpolator`, `EdgeInterpolator`, `Interpolant` are based on their counterpart in value analysis refinement. The interface `StrongestPostOperator` provides the functionality of the strongest-post operator in the refinement algorithms. Its previous counterpart in value analysis refinement is the `ValueAnalysisTransferRelation` that was used without an interface. `InterpolantManager` is an interface providing a previously not needed functionality: To be able to use an interface for interpolants, this interface is introduced to assume creating them at one point only, through injecting an interpolant manager in other classes of refinement. `ForgetfulState` is the interface for states used and created by the `StrongestPostOperator` and used by the `EdgeInterpolator`. It provides means for checking whether an element of the state is needed during interpolation and re-adding it, if it is.

Its type parameter **T** describes the type forgotten information is stored in. The method `ForgetfulState.forget(MemoryLocation)` returns the forgotten information as type **T** and it is used to remember the forgotten information, if necessary. Another type parameter we use is **S**, which represents the `ForgetfulState` implementation used. `Interpolant` and `FeasibilityChecker` don't need the additional functionality this interface provides, so we just use its super-type `AbstractState` for these. The third and last type parameter, **I**, describes the concrete `Interpolant` implementation used. The use of a type parameter describing a concrete implementation instead of just using `Interpolant<S>` everywhere allows implementing classes to use methods specific to certain implementations.

6.1.3 Creation of generic refinement classes based on refactoring of value analysis refinement

After introducing above interfaces, we create new generic refinement classes implementing these interfaces for the domain of abstract variable assignments by using and refactoring most of the code of the existing value analysis refinement. The resulting structure can be seen in the UML diagram of Figure 6.3. All aggregation relationships represent dependency injection through the constructor of the classes. All parts of the refinement procedure are easily interchangeable. `GenericRefiner` is an abstract class. By implementing the method `refineUsingInterpolants(ARGReachedSet, InterpolationTree)` that is expected to use an interpolation tree to update the precision and reset the reached and waitlist sets represented by the type `ARGReachedSet`, subtypes can represent a complete refinement procedure. It is possible to reuse all of the shown classes. One only has to implement an `Interpolant`, a `ForgetfulState`, an `InterpolantManager` managing this interpolant type and supporting the state type, and a `StrongestPostOperator` using the state type. These types then have to be injected either through the constructor of the `Generic*` classes, or by choosing the correct type parameter.

6.2 Refinement of Symbolic Value Analysis + Constraints CPA

Refinement of the symbolic value analysis CPA and constraints CPA is strongly based on these generic implementations. Besides `Interpolant`, `ForgetfulState`, `InterpolantManager` and `StrongestPostOperator`, we only create an own implementation of `EdgeInterpolator`. For all other components, we inherit the behaviour of the generic implementations. Figure 6.4 shows the structure of this refinement.

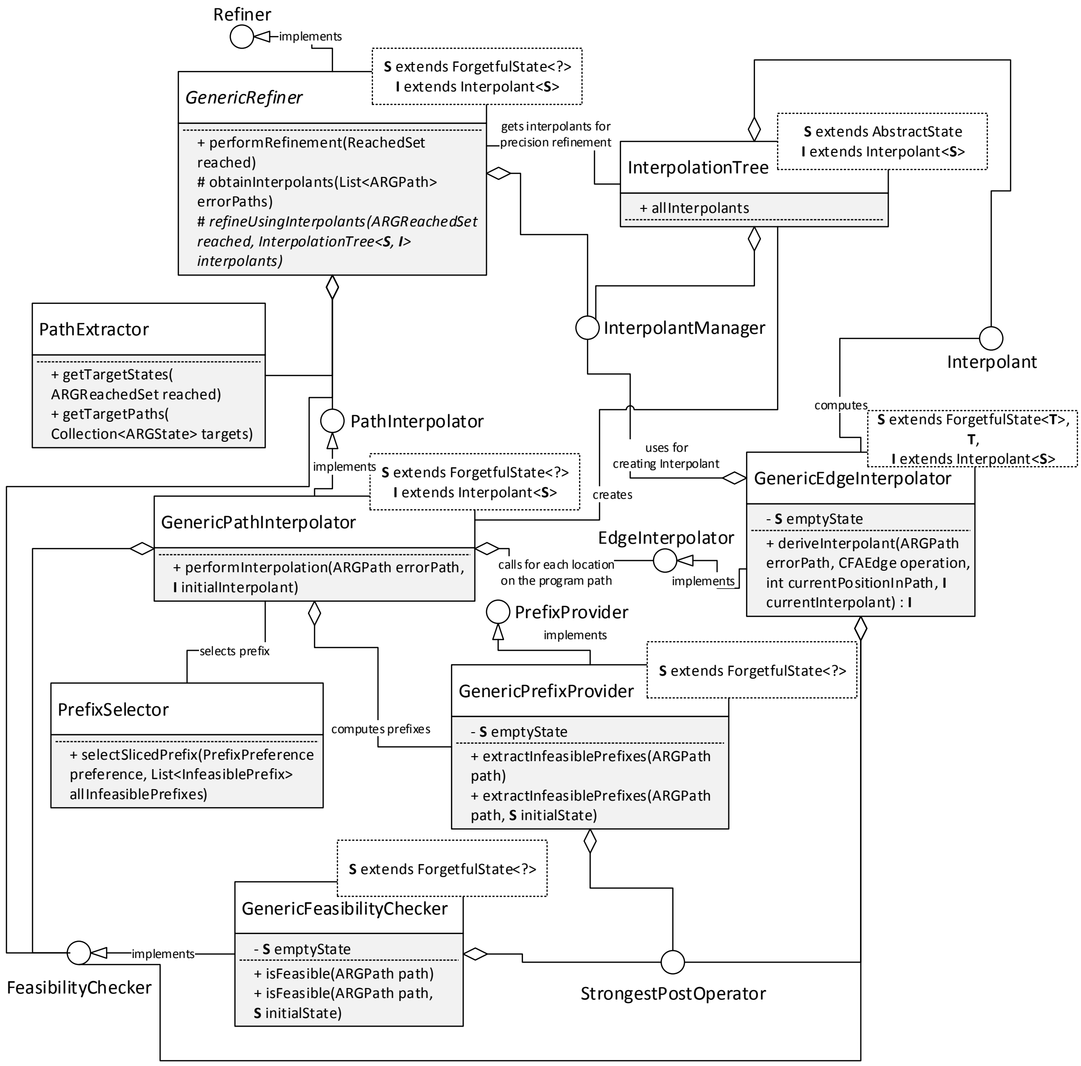


Figure 6.3: Structure of generic refinement procedure for abstract variable assignments

`SymbolicInterpolant` implements `Interpolant`. It stores information about abstract variable assignments and constraints, so it can be used for interpolating over both these types. `ForgettingCompositeState` implements `ForgetfulState`. It is the composition of `ValueAnalysisState` and `ConstraintsState` and provides methods for forgetting and remembering both their elements separately. This is necessary for interpolation, described below. `SymbolicInterpolantManager` is an `InterpolantManager` able to create `SymbolicInterpolants`. `Value-TransferBasedStrongestPostOperator` is the implementation of the compos-

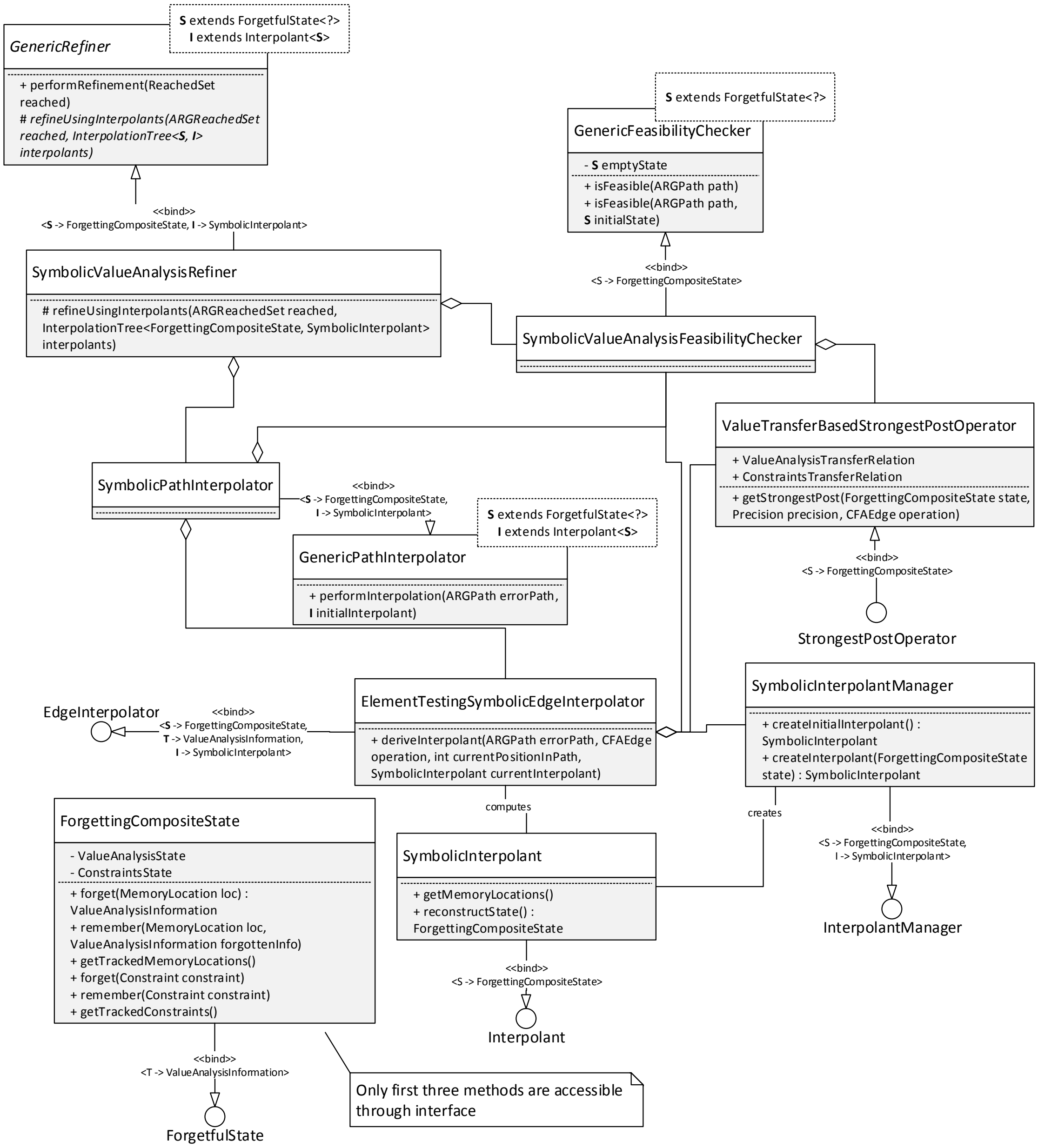


Figure 6.4: Structure of refinement procedure for symbolic execution

ite strongest-post operator using the value analysis transfer relation and constraints transfer relation, described in Section 4.3.1.

`SymbolicEdgeInterpolator` implements `EdgeInterpolator`. We can't use the functionality of `GenericEdgeInterpolator` since, depending on the configuration, we have to interpolate testing both constraints and/or variable assign-

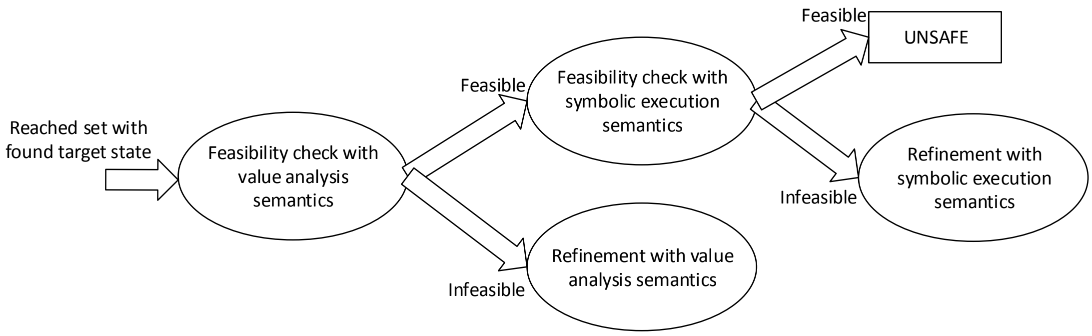


Figure 6.5: Symbolic execution refinement procedure. Before using constraints, try to prove infeasibility with value analysis semantics only

ments for their necessity. The generic edge interpolator only tests program variables (`MemoryLocations`), though.

6.2.1 Performing value analysis refinement first

The strongest-post operator SP^S of symbolic execution refinement performs a SAT check at every assume operation, just like the symbolic execution CPA's transfer relation. To minimize these expensive computations, we implement an additional refinement procedure that uses the semantics of value analysis's strongest-post operator SP only, if possible. To do this, we can't just use value analysis refinement because of the wrong interpolant type of Γ and the different set of precisions. If value analysis refinement was to update the precision by taking the current precisions of the locations whose states were removed from the reached set after successful interpolation and combining them with the newly computed precision, it would only consider the precision of the value analysis CPA and discard the existing precision of the constraints CPA. So instead, we build a new refinement procedure with a strongest-post operator representing the semantics of SP (see Section 3.3.2), generic refinement classes for interpolation and feasibility check using this strongest-post operator and `ForgettingCompositeState`, as well as `SymbolicValueAnalysisRefiner`, which considers the precisions of both value analysis CPA and constraints CPA.

When refining, we first call this procedure. If it is able to prove the error path as infeasible, we use its refined precision. If it is not, we use our refinement procedure for symbolic execution to get a new precision. This way we only use SAT checks in refinement and only increase the precision of the expensive constraints CPA if this is really necessary for computing an error path as infeasible.

6.2.2 Extract precision from predicate refinement

For using the refinement procedure extracting a precision from the predicate precision created by predicate CPA's refinement, predicate CPA's refinement is just executed and all program variables are extracted from the predicates of the resulting precision for each location. These program variables are then used for the precision of the value analysis CPA, while the locations are used as precision for the constraints CPA, since it is not possible to derive an original assume statement from the predicates created by predicate CPA's refinement. Since the predicate CPA is more powerful than the symbolic execution CPA, it is possible that a refined precision is returned that is not sufficient for the symbolic execution CPA to prove the infeasibility of the error path. Because of this it is not always possible to use this alternative.

7 Evaluation

7.1 Evaluation Setup

We performed each run of our benchmarks on a dedicated, unloaded server with an Intel Xeon E5-2650 v2 with 2.60 GHz and 32 CPU cores, using the Linux operating system Ubuntu 14.04 for the x86_64 architecture. The resource limits for each run were 15.00 GB (13.97 GiB) of memory (RAM), a Java heap limit of 10.49 GB (10000 MiB = 9.767 GiB), a maximum use of two CPU cores, a CPU time limit of 900 seconds (15 minutes), after which CPAchecker is supposed to shut down, and a hard time limit after which the run is killed of 1200 seconds (20 minutes). We chose a relatively big difference between the time limit in CPAchecker and the hard time limit to give the analysis enough time to shut down in case of a long taking SAT check during analysis. As SAT checks are delegated to an SMT solver, CPAchecker is unable to shut down during a check.

We took a subset of the benchmark repository[1] of the SV-COMP 2015 verification tasks for our benchmarks. A detailed explanation of all verification tasks present there can be found at [Bey15a]. We used:

1. BitVectors. Requires treatment of bit-operations, which allows us to check the need for and performance of a bitvector-based theory for SMT solving.

2. Floats. Requires handling of floats, which allows us to check the need for and performance of a float-based theory for SMT solving. An alternative is to just use rationals as approximations, whose use increases performance of SMT solving in comparison to floats.

3. ControlFlowInteger, ECA, Loops, ProductLines and Simple. All five of these sets are designed for testing the control flow and integer variable handling of analyses. Path explosion should occur here a lot.

[1] `https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp15`

4. DeviceDrivers. This set of tasks consists of problems that require analysis of pointer aliases and function pointers. Since the value analysis CPA can't handle pointers, we expect the symbolic execution CPA to perform poorly, also. This set uses a simple memory model and the 64-bit architecture, the only task category we use that does so.

5. HeapManipulation. This set of tasks also consists of problems that require analysis of pointer aliases and function pointers, as well as data structures on the heap. In contrast to the set DeviceDrivers, this set uses a precise memory model and a 32-bit architecture.

6. Sequentialized. Different tasks derived from SystemC programs. SystemC provides means to simulate concurrent processes. Such programs were transformed to pure C programs so they can be analyzed by CPACHECKER.

A simple memory model denotes that variables can only be modified using direct assignments or by using a pointer which was obtained by using & on the corresponding variable. A precise memory model denotes that all memory cells can be written to, even by dereferencing uninitialized pointers. We told CPACHECKER whether to assume a 32-bit or 64-bit architecture with the command-line parameters `-32` (actually the default) and `-64`.

The external method `__VERIFIER_nondet_X()` is used to introduce non-deterministic values of type `X` in a program. Although CPACHECKER is able to handle recursive function calls using block-abstraction memoization (BAM) [Fri15], we do not use this feature, but skip such calls to keep our analysis focused on the performance of the symbolic execution CPA and its comparisons. To skip recursive function calls, we use the command-line parameter `-skipRecursion`.

All verification tasks are batch executed using a benchmark script that performs all runs with above specifications and which returns for each run one of the following results:

- TRUE, if the program is safe, i.e. the specification holds
- FALSE, if the program is potentially unsafe, i.e. the analysis found a specification violation and can't prove that the specification holds due to this
- UNKNOWN, if the result is unknown, due to an error, a crash, or exhausted resources (e.g. time or memory)

For each such result, the script assigns a number of points depending on the received and the expected result, and presents the sum of these points as a general indicator for the performance of the used program/analyses. The points assigned are:

Points	Result
0	UNKNOWN
+1	FALSE, correct
-6	FALSE, incorrect (false alarm)
+2	TRUE, correct
-12	TRUE, incorrect (unsound analysis)

This point scale rewards correct results, but punishes wrong ones stronger, especially unsoundness, the worst property a verifier can have. The script rewards correctly found bugs with less points then proving that a specification holds, since the latter is more complicated. In addition, the script does not check whether the error found by the analysis is an actual specification violation or just a lucky coincidence due to a too high level of abstraction, when a correct FALSE is returned. That means that simple analyses like the value analysis CPA, which don't track much information, can get points for finding a potential bug that not really is one in case another real bug exists. Keeping these points in mind, the resulting score for an analysis can give a fair general overview of its performance.

All benchmarks were executed using the code of revision 17223 of the branch *symbolic-cegar* in the CPACHECKER repository[2]. All benchmark results can be found on the supplementary web page at `http://leostrakosch.github.io/symbolicValueAnalysis-enhanced/`.

7.2 Evaluation of CEGAR

7.2.1 Comparison to symbolic execution CPA without CEGAR

Evaluation shows the great boost CEGAR provides to the symbolic execution CPA. Table 7.1 shows the results of the symbolic execution CPA without CEGAR using the subset less-or-equal operator and merge^{sep} (SymEx w/o CEGAR), in comparison to the symbolic execution CPA using CEGAR with refinement based on CEGAR for explicit-state model checking (SymEx w/ CEGAR, Sec. 4.3.1). *Program errors*

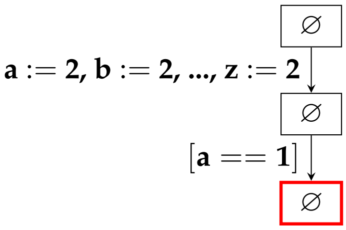
[2] `https://svn.sosy-lab.org/software/cpachecker/branches/symbolic-cegar`

	SymEx w/o CEGAR	SymEx w/ CEGAR	Overall
correct results	761 (18.60%)	2078 (50.78%)	4092
FALSE, correct	598 (50.63%)	376 (31.83%)	1181
TRUE, correct	163 (5.599%)	1702 (58.47%)	2911
unique FALSE, correct	323	101	
unique TRUE, correct	84	1623	
FALSE, incorrect	44	83	
unique FALSE, incorrect	4	43	
TRUE, incorrect	0	1	
program errors	2	2	
resource errors	3285	1928	

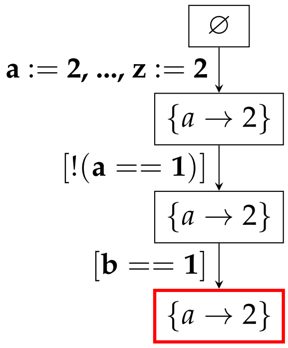
Table 7.1: Results of benchmark runs of the symbolic execution CPA without CEGAR and with CEGAR

are errors in the execution of CPACHECKER, in this case a parsing error of a file for both analysis and an exception due to a failure of the SMT solver in the analysis without CEGAR and one due to a division by zero in the analysis with CEGAR. The increase in correctly handled tasks *without* a safety violation (in the table row ″correct negatives") by a factor of more than 10 and the decrease in timeouts by more than 40% are the most notable improvements by using CEGAR. On the contrary, the number of found safety violations decreases by 222 tasks, since the lazy approach of CEGAR has a problem with programs consisting of a lot of assumptions leading to an error in dependence of many variables.

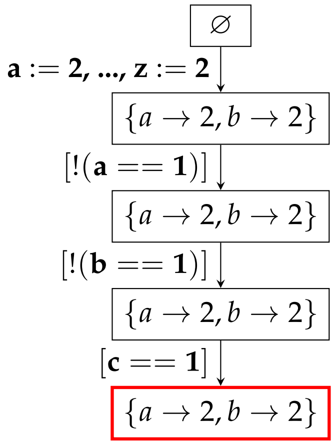
Figure 7.1d shows a CFA representing one such program. The highlighted nodes are error locations. Although only the last one of them is really reachable as all program variables are initialized with the concrete value 2 at the beginning of the program, the CEGAR algorithm visits one after the other, always refining the precision to track only one additional variable and then restarting from the beginning of the program, since all variable assignments happen there. The first three iterations of this procedure are shown in Figures 7.1a – 7.1c. This lazy approach performs many computations obviously unnecessary and as such has a significant worse performance than an eager approach using full precision. Using full precision, it is possible to prove all error paths but the last infeasible in one run, since the value analysis state already equals $\{a \rightarrow 2, b \rightarrow 2, ..., z \rightarrow 2\}$ after processing the first CFA edge (Fig. 7.1e). Analogous, programs like this requiring the tracking of all constraints exist and programs with such characteristics, but without a reachable



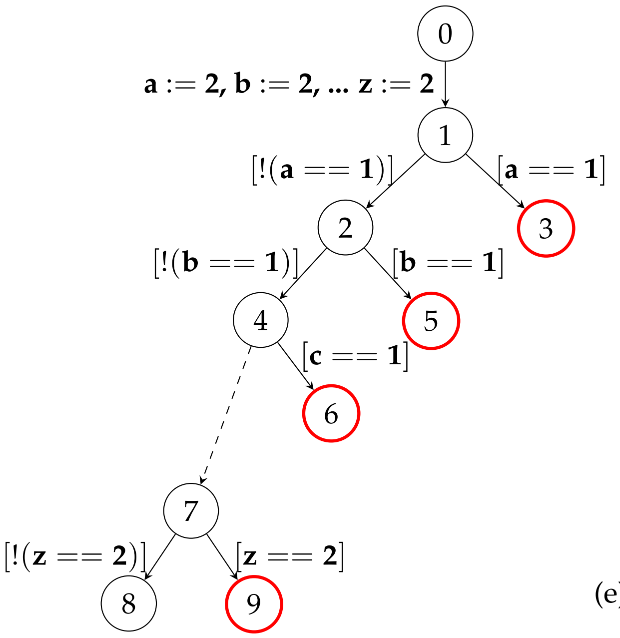
(a) First iteration, tracking no variables



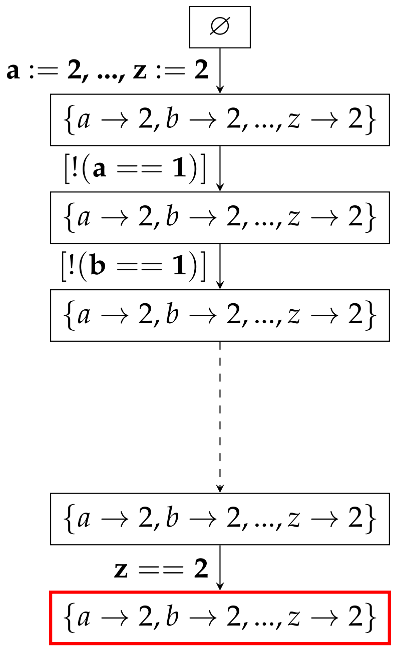
(b) Second iteration, tracking variable a



(c) Third iteration, tracking variables a and b

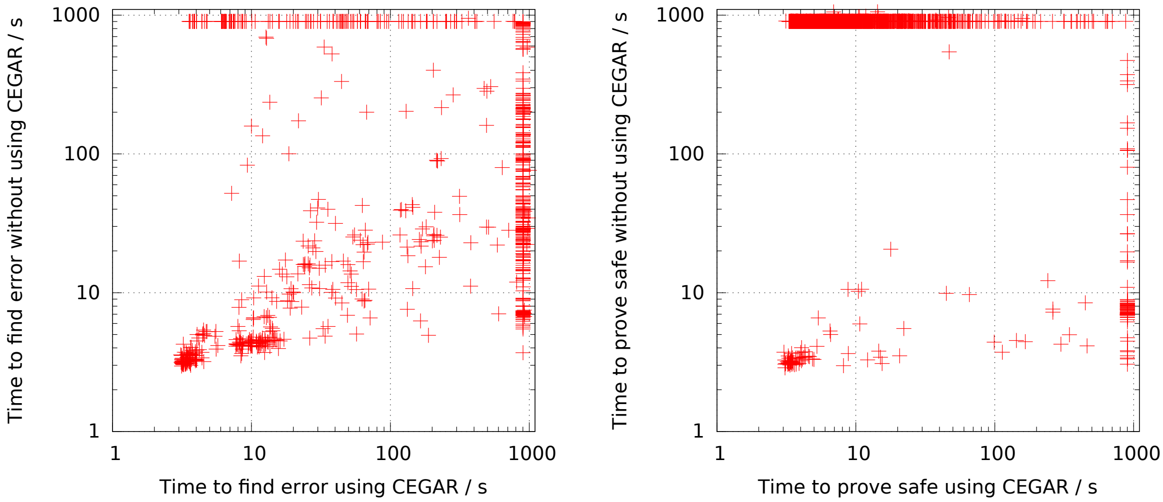


(d) CFA



(e) Last iteration tracking all program variables. Also equals the run of eager analysis

Figure 7.1: A CFA representing a program CEGAR performs worse for than eager analysis and the first three and last one iteration of analysis using CEGAR. The last iteration also equals the eager analysis



(a) Time in seconds to find an error

(b) Time in seconds to prove a program safe

Figure 7.2: Runtime performance of symbolic execution with and without CEGAR in comparison

target location. Tasks of the latter category constitute almost all of the 84 unique correct TRUE results of symbolic execution without CEGAR.

Most of the programs with these characteristics are of the task sets of ProductLines and ECA. To recap, of the 598 tasks symbolic execution without CEGAR can correctly find errors in, more than half (323 tasks) can't be analyzed correctly by symbolic execution with CEGAR due to many infeasible error paths and the resulting amount of refinements. On the other hand, symbolic execution with CEGAR is able to prove for 101 new tasks that an error exists in them. This shows that the efficiency of symbolic execution with and without CEGAR strongly depends on the task on hand, especially when the program is not error-free. Fig. 7.2a illustrates that for many programs, either symbolic execution with or without CEGAR is able to find a (possibly non-existent) error within 900 seconds, but not both. For proving the safety of a program, analysis with CEGAR performs significantly better, being able to correctly analyze 1623 tasks more than analysis without CEGAR, but its laziness results in bad performance for some programs, too (Fig. 7.2b).

For most tasks symbolic execution with CEGAR is able to compute a result but symbolic execution without CEGAR is not, only few or zero refinements are necessary (Fig. 7.3). Comparison with the random distribution of performed refinements in tasks that resulted in a timeout when using CEGAR (Fig. 7.4) confirms the problem CEGAR has with many possible error paths.

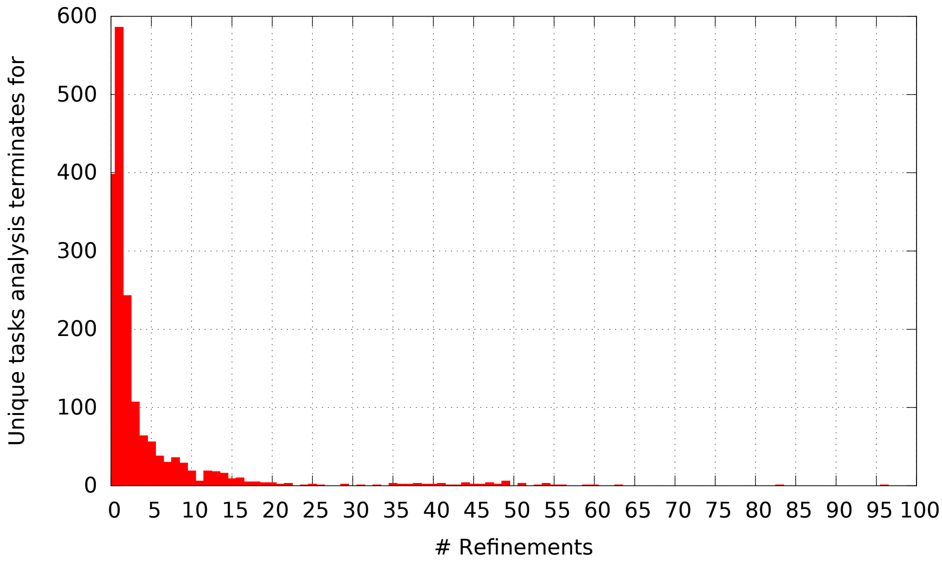


Figure 7.3: Amount of tasks analysis with CEGAR can compute a result for while analysis without CEGAR can't, and the number of refinements necessary for them

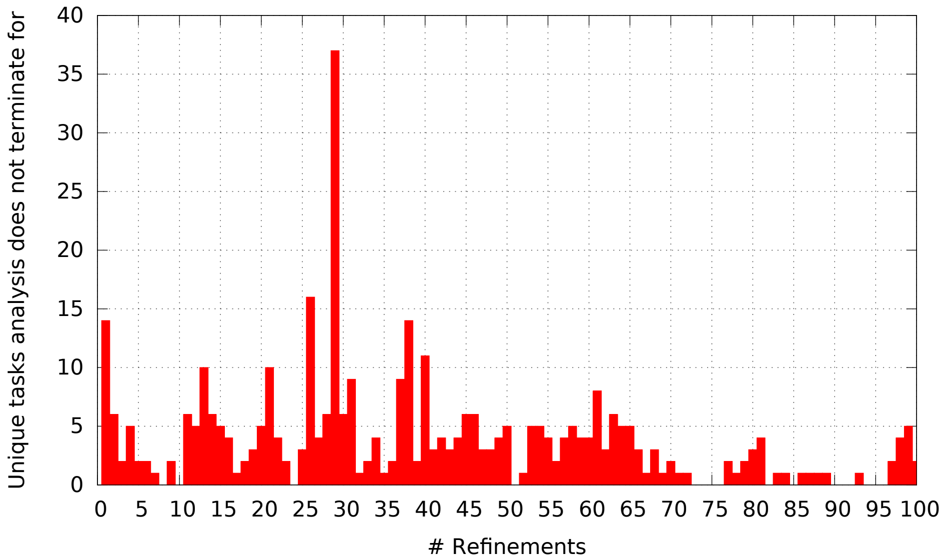


Figure 7.4: Amount of refinements (up to 100) that were performed for a specific amount of tasks that resulted in a timeout

Nevertheless, thanks to its significant better performance in proving the safety of programs, CEGAR was able to push the symbolic execution CPA's score from 660 points to 3271 points by increasing the amount of successfully verified error-free tasks by more than 1500.

	Values only	Values first	Constraints first	Overall
correct results	2080	2079	2078	4092
FALSE, correct	378	376	376	2911
TRUE, correct	1702	1703	1702	1181
unique FALSE, correct	2	0	0	
unique TRUE, correct	0	0	0	
FALSE, incorrect	83	83	83	
unique FALSE, incorrect	0	0	0	
TRUE, incorrect	1	1	1	
unique TRUE, incorrect	0	0	0	
program errors	3	3	2	
resource errors	1925	1926	1928	

Table 7.2: Results of benchmark runs of the symbolic execution CPA with CEGAR using three different techniques for interpolation computation

7.2.2 Interpolation techniques

We compare three different techniques for computing interpolants of the symbolic execution CPA already mentioned in Section 4.3.1:

a) Only removing unnecessary values and using all constraints that resulted from the previous interpolant and the strongest-post operator at the current edge (*values only*),

b) removing unnecessary values first and then constraints (*values first*), and

c) removing unnecessary constraints first and then values (*constraints first*).

Table 7.2 shows that almost no difference exists in the effectiveness of all three techniques. General time performance also differs in no significant way, as the scatter plots in Figure 7.5 show. Using the values only interpolation technique yields two more correctly found errors over all benchmark tasks. Both tasks are close to the time limit with 863.4 and 887.5 seconds. The successful analysis of these two tasks is a result of the faster "values only" interpolant computation. Since the strongest-post operator of symbolic execution refinement uses expensive SAT checks to check the satisfiability of current constraints, the two refinement procedures "values first" and "constraints first" take longer for a single refinement, as they call the strongest-post operator more often - for every abstract variable assignment and every constraint once, respectively. The "values only" computation only calls the strongest-post operator once for every value, in contrast. Similarly, analyses using "values only"

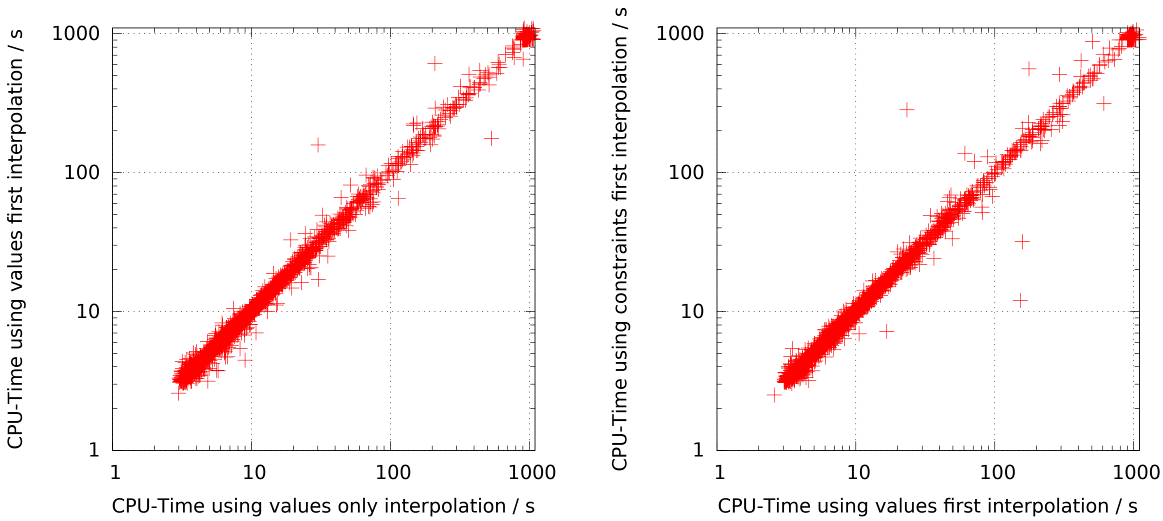


Figure 7.5: Runtime performance of different interpolation techniques over all benchmark tasks

and "values first" are able to prove one more task safe than "constraints first". This one is very close to the time limit (867.7 seconds using "values only", 836.6 seconds using "values first"), too.

Analysis with interpolation using "values first" or "constraints first" is also able to prove one task safe "values only" can't. Here, the computation times of 656.0 and 642.0 seconds are farther away from the time limit. Although refinement takes longer than with "values only", the coarser precision of the constraints CPA speeds up termination of the analysis after all information necessary for proving all error paths is tracked.

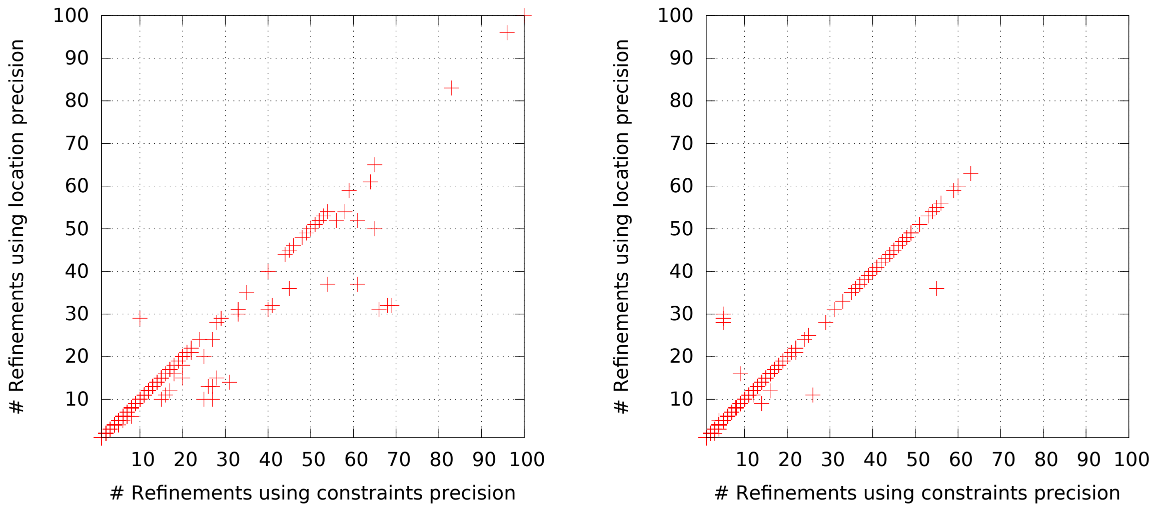
For further evaluation we will use the "constraints first" technique, as it is the closest to our specification and does not pose any disadvantages in comparison to the other two techniques.

7.2.3 Different sets of precision

Table 7.3 shows the performance of the symbolic execution CPA with CEGAR using the default constraints-based precision in comparison to the location-based precision. As it is already the case with the "value only" interpolation technique, using the location-based precision provides a small boost in the amount of correctly found property violations in exchange for a small decrease in tasks correctly proven safe. This is due to less needed refinements to reach a necessary precision to either find a feasible error path (Fig. 7.6a) or prove a program safe (Fig. 7.6b), as more constraints

	Constraint	Location	Overall
correct results	2078	2083	4092
FALSE, correct	376	384	2911
TRUE, correct	1702	1699	1181
unique FALSE, correct	1	3	
unique TRUE, correct	6	9	
FALSE, incorrect	83	83	
unique FALSE, incorrect	0	0	
TRUE, incorrect	1	1	
unique TRUE, incorrect	0	0	
program errors	2	2	
resource errors	1928	1923	

Table 7.3: Results of benchmark runs of the symbolic execution CPA with CEGAR using constraints-based and location-based precision for constraints CPA



(a) For finding an error

(b) For proving a program safe

Figure 7.6: Number of needed refinements for finding errors and proving a program safe

are tracked potentially. To prove a program safe, only few variables and constraints have to be tracked, most of the time, though. Because of this, the higher precision and the resulting bigger state space is often hindering when proving the safety of a program.

	Subset	Implication	Overall
correct results	2078	2078	4092
FALSE, correct	376	376	2911
TRUE, correct	1702	1702	1181
unique FALSE, correct	1	1	
unique TRUE, correct	0	0	
FALSE, incorrect	83	83	
unique FALSE, incorrect	0	0	
TRUE, incorrect	1	1	
unique TRUE, incorrect	0	0	
program errors	2	2	
resource errors	1928	1928	

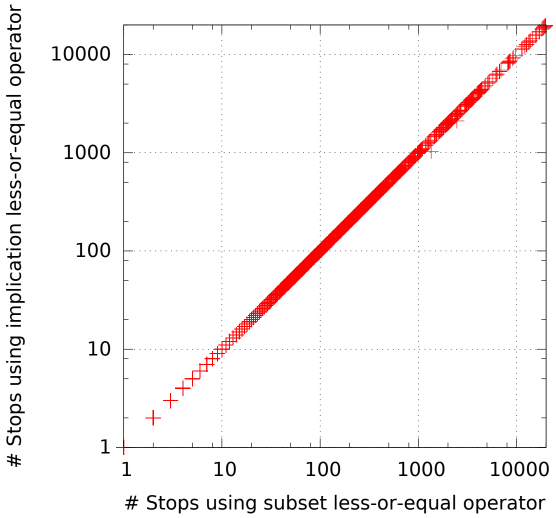
Table 7.4: Results of benchmarks runs of the symbolic execution CPA with CEGAR using the subset and the implication less-or-equal operator with the constraints-based precision for the constraints CPA

	SL	IL	Overall
correct results	2083	2079	4092
FALSE, correct	384	384	2911
TRUE, correct	1699	1695	1181
unique FALSE, correct	1	1	
unique TRUE, correct	4	0	
FALSE, incorrect	83	83	
unique FALSE, incorrect	0	0	
TRUE, incorrect	1	1	
unique TRUE, incorrect	0	0	
program errors	2	3	
resource errors	1923	1926	

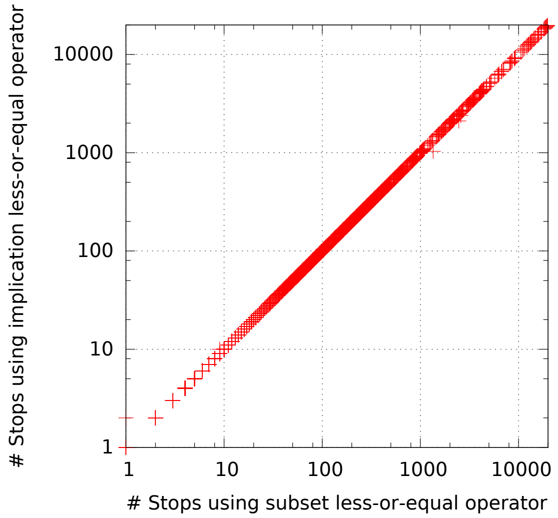
Table 7.5: Results of benchmark runs of the symbolic execution CPA with CEGAR using subset (SL) and implication (IL) less-or-equal operator with the location-based precision for constraints CPA

7.2.4 Less-or-equal operators

The implication less-or-equal operator never performs better than the subset less-or-equal operator when using the constraints-based precision for the constraints CPA and worse than the subset less-or-equal operator when using the location-based precision. Tables 7.4 and 7.5 show the results for benchmarks using these two different less-or-equal operators. The reason for this is the fixedness of constraints: The possibility that a state using a subset of or the same symbolic identifiers than



(a) Using the constraint-based precision



(b) Using the location-based precision

Figure 7.7: Number of times stopped with subset and implication less-or-equal operator

another state consists of constraints that are implied by the other state without being a subset of its constraints, is very low. Figure 7.7 illustrates that this is the case very seldom. As such, usage of the subset less-or-equal operator is encouraged due to its higher simplicity and no reliance on third-party SMT solvers.

7.2.5 Sliced prefix selection

The use of sliced prefix selection [BLW15b] [BLW15a] (Sec. 3.3.2) is able to boost the performance of the symbolic execution CPA with CEGAR significantly. Table 7.6 shows the benchmark results of the following selected sliced prefix selection preferences:

- Random sliced prefix selection, used as reference selection preference. item Selection of the shortest prefix, *length short*.
- Selection of the prefix based on a score computed from the variable types, easy types like boolean and integer being preferred. (domain good, *DG*)
- Selection of the prefix based on a score computed from the variable types equal to "DG", but mixed with a score based on the size of the interpolants, preferring smaller ones. (domain good, narrow, *DGN*)
- Selection of the prefix containing the fewest assignments, *AmF*.

Pref.Selection	FALSE, corr.	TRUE, corr.	FALSE, incorr.	TRUE, incorr.	Score
Random	488	1806	88	1	3560
Length short	454	1738	93	1	3366
DG short	476	2000	94	1	3906
DG long	466	1949	93	1	3800
DGN short	482	1988	93	1	3888
AmF short	459	1688	82	1	3331
AmF long	407	1691	84	1	3279
AtF short	475	1624	93	1	3159
AtF long	412	1672	82	1	3258
AtM short	522	1843	93	1	3638
AtM long	391	1827	92	1	3487
PS short	447	1812	94	1	3501
PS long	516	1810	88	1	3602

Table 7.6: Benchmark results for different sliced prefix selection types, best and worst results highlighted

- Selection of the prefix containing the fewest assumptions, *AtF*.
- Selection of the prefix containing the most assumptions, *AtM*.
- Selection of the prefix closest to the initial location of the error path. (pivot shallow, *PS*)

For all preferences but the random one a second preference exists if multiple prefixes are equal in concern to the preference: *short* for choosing the shortest prefix with the best score and *long* for choosing the longest prefix with the best score. More information about the individual prefix preferences can be found in [BLW15a].

It is clearly visible that performance of analysis strongly depends on the type of interpolants used for refinement, with symbolic execution being able to prove the safety of 366 more tasks when using the sliced prefix selection preference *domain good, short* in contrast to *assumptions fewest, short*.

Using preferences aiming at increasing precision fast like *assumptions most, short* or *pivot shallow, long* allows analysis to find more errors thanks to its faster-growing precision and fewer needed refinements, just as expected. Preference *assumptions most, short* increases the precision of the constraints CPA as fast as possible by always incrementing the amount of constraints tracked as much as possible by choosing sliced prefixes relying on the most assumptions for proving infeasibility of the prefix. Preference *pivot shallow, long*, chooses the prefixes closest to the initial location of the

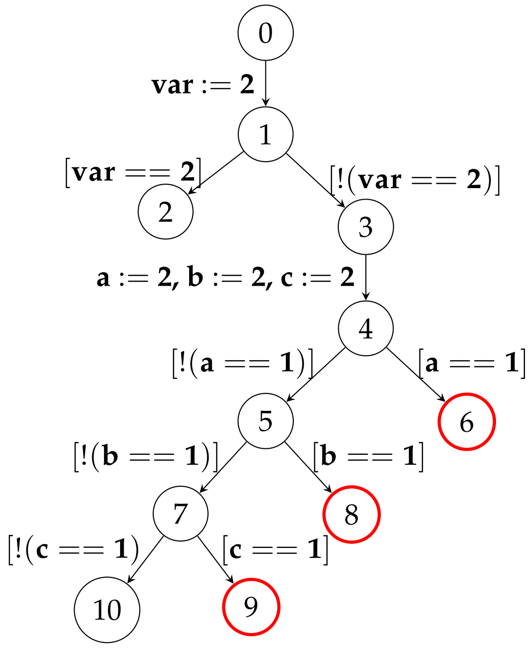
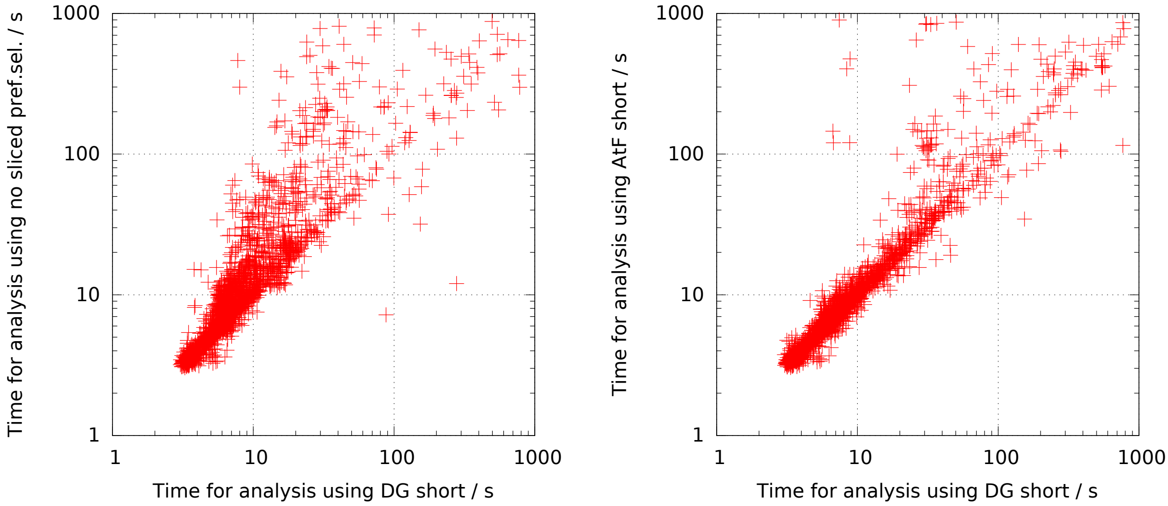


Figure 7.8: CFA representing a program that creates problems when using the wrong sliced prefix selection preference

error path so that new precisions are propagated to the most abstract states possible. It takes the longest prefixes, in addition, so that precision grows fast. Increasing precision fast and continuing analysis after refinement higher in the abstract reachability graph avoids unnecessary refinements for error paths that are infeasible for the same reasons. An example CFA for this benefit can be seen in Fig. 7.8. While it is similar to the CFA in Fig. 7.1d, expensive refinement procedures can be avoided by choosing to track program variable *var*. If the prefix preference always chooses sliced prefixes close to the target location, three refinement procedures are necessary at the end of which program variables *a*, *b* and *c* are tracked. If the prefix preference chooses the sliced prefix using program variable *var*, analysis already terminates after one refinement.

In contrast to these, *assumptions fewest, short* has the slowest growing precision, increasing precision of the constraints CPA only as little as possible by choosing the fewest assumptions possible and precision of the symbolic value analysis CPA only as much as needed to keep constraints CPA's precision low by choosing short prefixes. The slow growth of precision is also the cause *domain good, short* only performs mediocre in relation to the other analysis in finding errors. Thanks to choosing variables whose types are easily processible by both the value analysis CPA and the constraints CPA, it boosts proving the safety of programs immensely



(a) DG short vs. no sliced prefix selection

(b) DG short vs. AtF short

Figure 7.9: Comparison of runtime performance of analysis using *domain good, short* prefix preference with analysis without sliced prefix selection and with preference *assumptions fewest, short*

by improving runtime performance. Figure 7.9 shows two scatter plots comparing the claimed CPU-time of the symbolic execution CPA with CEGAR using the *domain good, short* preference in comparison to no sliced prefix selection and the relatively bad performing *assumptions fewest, short* preference. Only tasks are shown for which both analyses terminated and for which the same results were computed. It is clearly visible that analysis is faster for the significant amount of tasks when preferring precision refinements using easy-to-handle variables.

For the domain-good preference, a more precise alternative exists. It performed worse than the default for all variations, though.

It must be added that the benchmarks for sliced prefix preferences were run with configuration option `cpa.value.optimizeBooleanVariables=true`, which causes a bug for few tasks in the symbolic execution CPA. When using *domain good, short*, one task was affected by this error, which exceeds the time limit with the option turned off. For *assumptions fewest, short* and *pivot shallow, long*, no task was affected. For *assumptions most, short*, 5 tasks were affected. It still is the best preference for finding errors and the fourth-best for proving the safety of programs. Unfortunately there was no time to repeat all benchmarks with this option turned off due to their long runtime.

	No delegation	Delegation	Overall
correct results	2078	2081	4092
FALSE, correct	376	423	2911
TRUE, correct	1702	1658	1181
unique FALSE, correct	64	20	
unique TRUE, correct	86	89	
FALSE, incorrect	83	84	
unique FALSE, incorrect	2	2	
TRUE, incorrect	1	1	
unique TRUE, incorrect	0	0	
program errors	2	3	
resource errors	1928	1924	

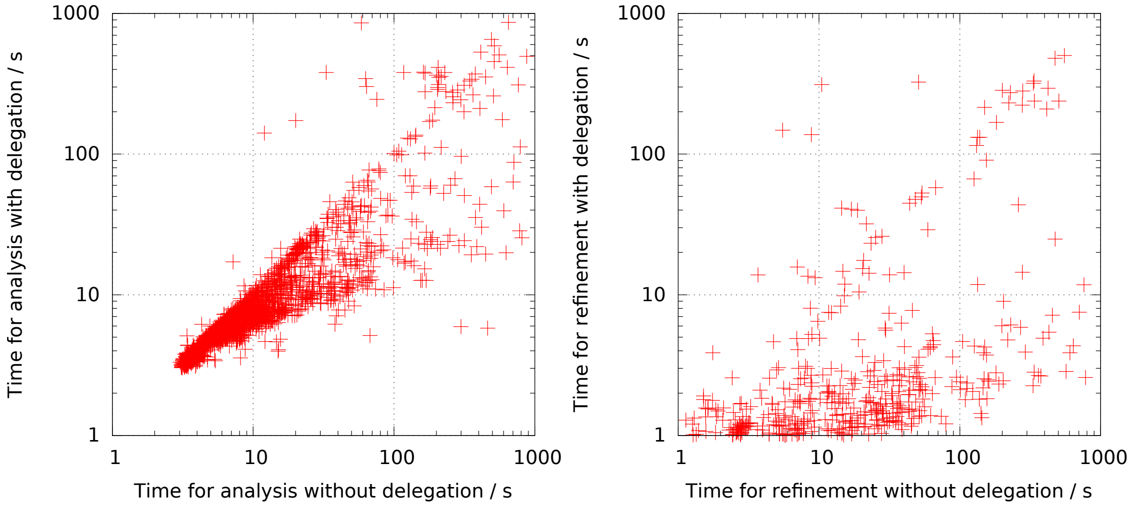
Table 7.7: Results of benchmark runs of the symbolic execution CPA with CEGAR using no delegation (the default) and using delegation to concrete value analysis only refinement, both using no sliced prefix selection

7.2.6 Delegation to concrete value analysis refinement

Without sliced prefix selection Table 7.7 shows the difference between analysis using CEGAR without and with delegation to refinement using concrete values only and no sliced prefix selection. While sometimes analysis always using symbolic refinement (using no delegation) performs better and sometimes analysis performing concrete value analysis whenever possible performs better, 137 of the 184 differences appears in the ECA task set, where the kind of refinement is very important.

Although both techniques differ slightly in the tasks they can solve, delegation provides a significant boost to speed for tasks they can solve both (which are almost all of the tasks they can solve, after all). Figure 7.10a shows the CPU-time both analysis with CEGAR using no delegation and analysis with CEGAR using delegation claim for tasks they both terminate for with the same result (no matter whether it is correct or incorrect). While there are some tasks analysis with delegation can solve faster using symbolic refinement all the time, it is clearly visible that for the majority of tasks, analysis using delegation performs significantly better. This is attributed to the faster refinement procedure that goes without expensive SAT checks, in contrast to symbolic refinement. Figure 7.10b illustrates this difference in speed.

[3] A new bug was discovered during this benchmark, affecting 10 tasks. Unfortunately, the bug could not be resolved in the scope of this work.



(a) CPU-time for complete analysis (b) CPU-time for refinement procedure

Figure 7.10: Performance of analysis using CEGAR without delegation and with delegation for tasks both techniques terminate for with the same result

	No deleg. + DG, short	Deleg. + DG, short	Overall
correct results	2476	2417	4092
FALSE, correct	476	462	2911
TRUE, correct	2000	1955	1181
unique FALSE, correct	24	10	
unique TRUE, correct	52	7	
FALSE, incorrect	94	86	
unique FALSE, incorrect	7	0	
TRUE, incorrect	1	1	
unique TRUE, incorrect	0	0	
program errors	2	12[3]	
resource errors	1519	1924	

Table 7.8: Results of benchmark runs of the symbolic execution CPA with CEGAR using no delegation (the default) and using delegation to value analysis refinement, both using no sliced prefix selection

With sliced prefix selection Sliced prefix selection improves performance of analysis using delegation in the same way it improves analysis without delegation (Table 7.8). Interestingly, the performance increase by the delegation disappears by the preferred use of easy-to-handle variables. Figure 7.11 illustrates this change. Of the 2487 tasks that are solved with the same result by the two analyses, 551 tasks are more than 10% faster using no delegation, while only 77 tasks are more than 10% faster using delegation. As a conclusion, not delegating to the value analysis

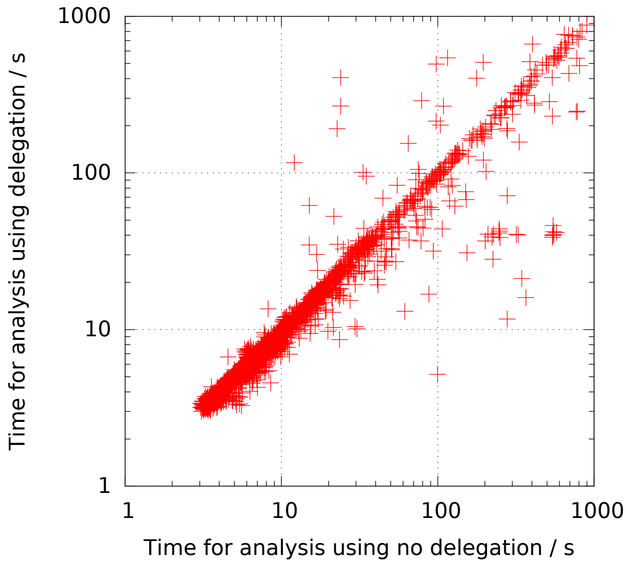


Figure 7.11: CPU-time claimed by analysis using *domain good, short* with and without delegation, for tasks the same result was computed for

	Own refinement	Predicate refinement	Overall
correct results	2476	2209	4092
FALSE, correct	476	384	2911
TRUE, correct	2000	1825	1181
unique FALSE, correct	133	41	
unique TRUE, correct	187	12	
FALSE, incorrect	93	12	
unique FALSE, incorrect	7	5	
TRUE, incorrect	1	0	
unique TRUE, incorrect	0	0	
program errors	2	363	
resource errors	1519	1508	

Table 7.9: Results of benchmark runs of the symbolic execution CPA with CEGAR using our own refinement procedure and using the refinement procedure deriving precisions from predicate analysis's refinement.

refinement yields better results both in effectiveness and performance when using sliced prefix selection, which is encouraged.

7.2.7 Utilization of predicate refinement

Evaluation shows that the predicate CPA often creates predicate precisions that can't be used to derive precisions for the symbolic execution CPA. In some cases, however,

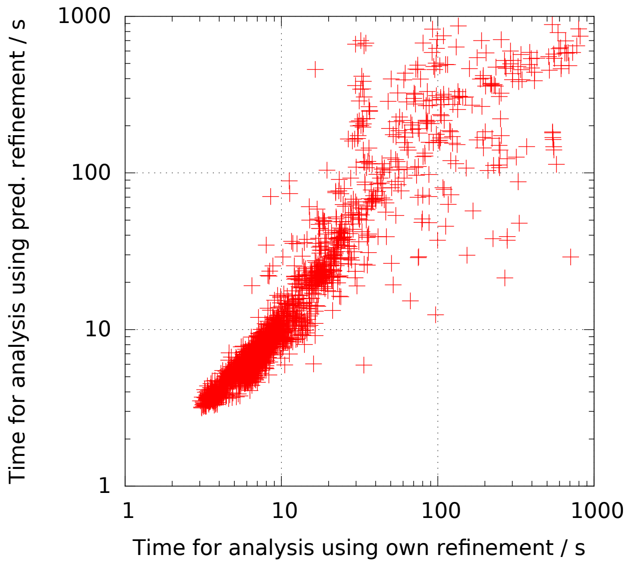


Figure 7.12: CPU-time claimed by analysis using symbolic execution with precisions created by symbolic execution refinement using sliced prefix selection and preference *domain good, short* and with precisions derived from predicate CPA's refinement

precisions are created that are more reliable and result in analysis terminating when our own refinement can't. Performance-wise, our own refinement is faster, either (Fig. 7.12). It might be useful to extend this approach to see if performance changes, but evaluation at the current point does not show much advantages compared to our dedicated refinement procedure.

7.3 Comparison to other CPAs

7.3.1 Value analysis CPA

Comparison to the value analysis CPA shows the strengths and weaknesses of symbolic execution, persisting even when using CEGAR: Due to its higher precision, its computation is more reliable in comparison to the more abstract value analysis, finding 200 less non-existent errors. But die to its higher precision, it also exceeds the time limit 535 times more often.

Of the 24 tasks symbolic execution computes correctly while value analysis does not, 13 are due to its precise handling of bitvectors and floats. 11 tasks are due to the handling of non-deterministic values. For all 24 of them, value analysis computes

	Value analysis	Symbolic execution	Overall
correct results	2814	2476	4092
FALSE, correct	798	476	2911
TRUE, correct	2016	2000	1181
unique FALSE, correct	322	0	
unique TRUE, correct	40	24	
FALSE, incorrect	294	94	
unique FALSE, incorrect	201	0	
TRUE, incorrect	0	1	
unique TRUE, incorrect	0	1	
program errors	1	3	
resource errors	983	1518	

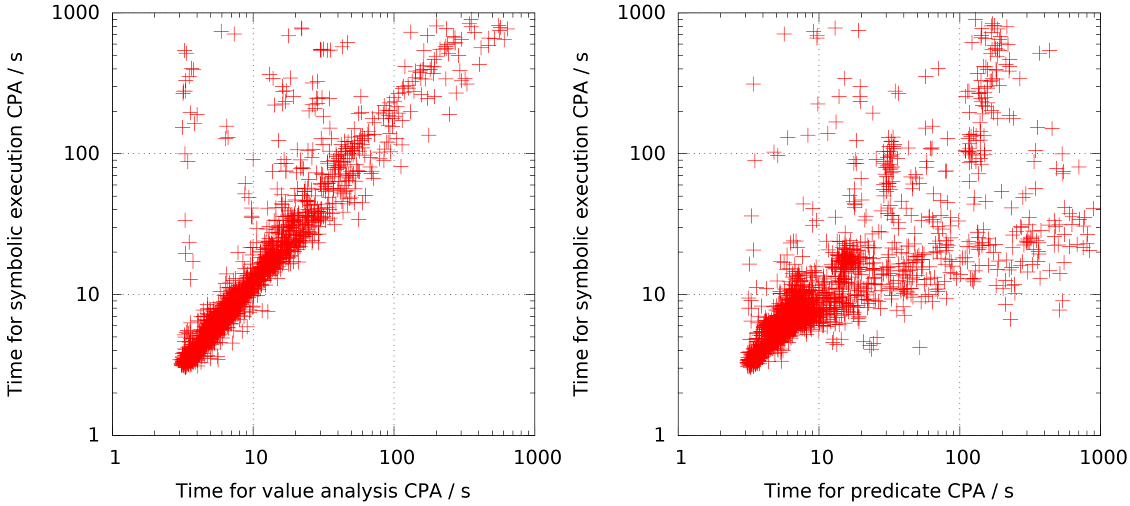
Table 7.10: Results of benchmarks of the value analysis CPA and the symbolic execution CPA, both using CEGAR with sliced prefix selection and the *domain good, short* preference

an incorrect result. The other 177 tasks analyzed incorrectly by the value analysis, symbolic execution exceeds the time limit.

Figure 7.13a displays the advantage in speed the value analysis CPA offers in comparison to the symbolic execution CPA. This difference was expected due to the higher precision of the symbolic execution CPA and expensive SAT checks in transfer relation and refinement. The overall score of symbolic execution is 3906 points, of value analysis 3066 points due to the higher amount of incorrect results.

7.3.2 Predicate CPA

For comparison to predicate analysis, we used the predicate CPA using bitvector and float theories with its default configuration, using adjustable-block encoding [BKW10]. The sophisticated and evolved predicate CPA is able to outperform symbolic execution in both finding errors and proving programs safe. It also computes less incorrect results. Nevertheless, symbolic execution using CEGAR is almost on par with predicate analysis for proving programs safe with only 57 tasks difference. The symbolic execution CPA is able to find errors for 118 programs the predicate CPA can't and prove the safety of programs for 144 programs the predicate CPA can't. Performance-wise, they differ greatly depending on the task, but with none of them being distinctively better than the other (Fig. 7.13b). In conclusion, the predicate CPA reaches a score of 4572 points in comparison to the symbolic execution CPA's 3906 points.



(a) Comp. with value analysis CPA (b) Comp. with predicate CPA

Figure 7.13: Comparison of CPU-time for value analysis CPA and predicate CPA with symbolic execution CPA for tasks each pair computes the same result for

	Predicate analysis	Symbolic execution	Overall
correct results	2677	2476	4092
FALSE, correct	620	476	2911
TRUE, correct	2057	2000	1181
unique FALSE, correct	262	118	
unique TRUE, correct	201	144	
FALSE, incorrect	21	94	
unique FALSE, incorrect	9	81	
TRUE, incorrect	3	1	
unique TRUE, incorrect	3	1	
program errors	20	3	
resource errors	1317	1518	

Table 7.11: Results of benchmarks of the predicate CPA and the symbolic execution CPA, both using CEGAR with sliced prefix selection and the *domain good, short* preference

This shows symbolic execution's potential for software verification, ranked between the value analysis CPA and the predicate CPA in both effectiveness, precision, and performance.

7.3.3 Comparison to TRACER

We tried to compare our symbolic execution approach using CEGAR to TRACER [JMNS12], a tool for software verification using eager symbolic execution with interpolation. Unfortunately, TRACER can't handle preprocessed files as they consist in the SV-COMP task sets, by default. Further investment in making these two tools comparable was not in the scope of this work. Of the 12 tasks evaluated in [JMNS12], all but one can be solved by the symbolic execution CPA correctly. TRACER was able to analyze 5 correctly when using strongest-post conditions for interpolation and all of them when using weakest-pre conditions. Since no information about the evaluation environment is given, it is not reliable to compare the runtime of the analyses.

8 Future work

Multiple improvements can be made to the symbolic execution CPA: To supplement symbolic execution with or without CEGAR, adding the handling of unbounded loops should be one of the biggest concerns. Multiple approaches were already mentioned in Section 2.

We haven't evaluated the use of symbolic execution with a simpler SMT theory like integer values and rationals instead of bitvectors and floats, yet. This should yield a big performance boost for SAT checks without creating any disadvantages in most programs, as only few rely on precise bitvector and float arithmetic in regards to their specification.

For improving symbolic executions competency in finding errors, two major approaches exist: Firstly, symbolic execution without CEGAR could be developed further by simplifying constraints states in different ways, reordering symbolic expressions by a fixed scheme, if possible, partially evaluating symbolic expressions as far as possible, and deleting constraints covered by others (e.g. $\{s1 > 0, s1 > 10\}$ could be simplified to $\{s1 > 10\}$). This way, termination checks would be able to reflect coverage of concrete states more precisely.

Secondly, symbolic execution using CEGAR could be improved further. We already showed the great benefits good selection preferences can produce in our evaluation. New sliced prefix selection preferences aimed at symbolic execution, specifically, could be developed. Not all existing selection preferences were evaluated in this work, either. In addition, functionality of symbolic execution could be altered to converge towards predicate abstraction. Instead of tracking constraints as they are derived from assumptions and performing the rather unsophisticated interpolation of just deleting constraints and testing whether they are needed for contradicting the suffix, interpolation could be performed by a SMT solver, creating more abstract interpolants. Constraints could then be combined from these interpolants' predicates, just like predicate abstraction does. In contrast to the latter, symbolic execution would do this for assumptions, only. This way, more abstract but uniform constraints states would be created, resulting in a higher hit rate for termination checks.

A third possibility for increasing the performance of CEGAR would be to introduce adjustable block-encoding [BKW10] to the constraints CPA, only performing abstraction at loop-headers, for example. This way, usage of a merge operator other than $merge^{sep}$ would be possible inside of blocks for the constraints CPA.

Symbolic execution's performance in general could be increased by applying the idea of concolic testing to symbolic execution in the context of software verification. When handling an assume edge in the constraints CPA, an assignment of symbolic identifiers to concrete values satisfying the current constraints could be stored in the constraints state and used for further checks. Only if the assignments do not fulfill a new constraint, a new SAT check is performed, computing a new satisfying assignment additionally, if one exists. This should hold potential to speeding up the handling of assume edges in the constraints CPA.

Search heuristics could also be used to guide analysis to target locations faster. Some heuristics already exist in CPACHECKER, for example different traversal strategies and the possibility to handle more abstract abstract states of the value analysis CPA, first. These heuristics could be evaluated and new could be added, taking inspiration from heuristics proposed in [BS08] and [CDE08].

A characteristic important for test generation and general usability is the correct creation and output of counterexamples. This is not supported by the symbolic execution CPA's refinement procedure, yet. A peculiarity of counterexample creation with symbolic execution is denoting and handling symbolic identifiers in the counterexample. For each symbolic identifier, a concrete value satisfying the counterexample could be computed and delivered as part or along of the counterexample. Alternatively, for each symbolic identifier the range of satisfying values could be computed, also more costly.

9 Conclusion

We successfully designed and implemented a refinement procedure to use symbolic execution with CEGAR in the context of configurable software verification. To our knowledge, we were the first to apply CEGAR to symbolic execution. In addition, we applied CEGAR not to one single domain, but to two strongly intertwined domains at once, by combining the symbolic value analysis CPA and the constraints CPA to represent the semantics of symbolic execution. As a second and to our knowledge novel approach we utilized the refinement procedure of another domain, namely predicate analysis, to derive a precision for our symbolic execution domain. Both approaches to refinement yielded comparable results. We evaluated two different less-or-equal operators and two different sets of precisions for the constraints CPA and illustrated their slight differences.

Additionally, we refactored value analysis CPA implementation's refinement procedure to be more generic. The now generic refinement procedure allows the implementation of new refinement procedures with low effort, in contrast to the previous need for a full implementation. This also allowed us to apply advanced features like sliced prefix selection to our new refinement procedure without any additional changes.

Evaluation shows the competitive performance of symbolic execution, being more reliable than value analysis and almost as effective as predicate analysis. Since it is possible to explicitly distinguish between concrete and symbolic (i.e. non-deterministic) values, symbolic execution might prove useful for test generation, too. Symbolic execution in the context of configurable software verification and especially in combination with CEGAR shows potential for software verification that should be build on to be able to verify even more programs.

Bibliography

[AGT08] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. *Demand-Driven Compositional Symbolic Execution*. Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 4963 of *LNCS*, pages 368–381. Springer-Verlag, 2008.

[Bey13] Dirk Beyer. *Second Competition on Software Verification (Summary of SV-COMP 2013)*. Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 7795 of *LNCS*, pages 594–609. Springer-Verlag, 2013.

[Bey14] Dirk Beyer. *Status Report on Software Verification (Competition Summary SV-COMP 2014)*. Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 8413 of *LNCS*, pages 373–388. Springer-Verlag, 2014.

[Bey15a] Dirk Beyer. *Competition on Software Verification (SV-COMP) 2015: Benchmark Verification Tasks*, 2015. Online available at `http://sv-comp.sosy-lab.org/2015/benchmarks.php`.

[Bey15b] Dirk Beyer. *Software Verification and Verifiable Witnesses (Report on SV-COMP 2015)*. Proceedings of the 21th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 9035 of *LNCS*. Springer-Verlag, 2015.

[BGS] Dirk Beyer, Sumit Gulwani, and David A Schmidt. *Combining Model Checking and Data-Flow Analysis*. To be released.

[BHT07] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. *Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis*. volume 4590 of *LNCS*, pages 504–518. Springer-Verlag, 2007.

[BHT08] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. *Program Analysis with Dynamic Precision Adjustment*. Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 29–38. IEEE Computer Society Press, 2008.

[BK11] Dirk Beyer and M. Erkan Keremoglu. *CPAchecker: A Tool for Configurable Software Verification*. Proceedings of the 23rd International Conference on Computer Aided Verification (CAV), volume 6806 of *LNCS*, pages 184–190. Springer-Verlag, 2011.

[BKW10] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. *Predicate Abstraction with Adjustable-Block Encoding*. Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD), pages 189–197. FMCAD, 2010.

[BL13] Dirk Beyer and Stefan Löwe. *Explicit-State Software Model Checking Based on CEGAR and Interpolation*. Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE), volume 7793 of *LNCS*, pages 146–162. Springer-Verlag, 2013.

[BLW15a] D. Beyer, S. Löwe, and P. Wendler. *Refinement Selection*. Proceedings of the 22nd International Symposium on Model Checking of Software (SPIN), LNCS. Springer-Verlag, 2015. To be released.

[BLW15b] Dirk Beyer, Stefan Löwe, and Philipp Wendler. *Sliced Path Prefixes: An Effective Method to Enable Refinement Selection*. Proceedings of the 10th International Conference on Distributed Computing Techniques (FORTE), volume 9039 of *LNCS*, pages 228–243. Springer-Verlag, 2015.

[BPR02] Tom Ball, Andreas Podelski, and Sriram K. Rajamani. *Boolean and Cartesian Abstraction for Model Checking C Programs*. Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 2031 of *LNCS*, pages 268–283. Springer-Verlag, 2002.

[BS08] J Burnim and K Sen. *Heuristics for Scalable Dynamic Test Generation*. Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 443–446. IEEE Computer Society, 2008.

[BW12] Dirk Beyer and Philipp Wendler. *Algorithms for Software Model Checking: Predicate Abstraction vs. IMPACT*. Proceedings of the 12th International Conference on Formal Methods in Computer-Aided Design (FMCAD), pages 106–113. FMCAD, 2012.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*. Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 209–224. USENIX Association, 2008.

[CGJ+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. *Counterexample-guided Abstraction Refinement for Symbolic Model Checking*. Journal of the ACM, volume 50, number 5, pages 752–794. 2003.

[CJM14] Duc-hiep Chu, Joxan Jaffar, and Vijayaraghavan Murali. *Lazy Symbolic Execution for Enhanced Learning*. Proceedings of the 5th International Conference on Runtime Verification (RV), volume 8734 of *LNCS*, pages 323–339. Springer-Verlag, 2014.

[Cra57] William Craig. *Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem*. The Journal of Symbolic Logic, volume 22, number 3, pages 250–268. 1957.

[Fri15] Karlheinz Friedberger. *Block-Abstraction Memoization as an Approach to Verify Recursive Procedures*. Master's thesis, Chair of Software Systems, University of Passau, 2015.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. *DART: Directed Automated Random Testing*. Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM, 2005.

[God07] Patrice Godefroid. *Compositional Dynamic Test Generation*. Proceedings of the 34th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pages 47–54. ACM, 2007.

[HJMM04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. *Abstractions from Proofs*. Proceedings of the 31st SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pages 232–244. ACM, 2004.

[HMS02] Thomas A. Henzinger, Rupak Majumdar, and Gregoire Sutre. *Lazy Abstraction*. Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pages 58–70. ACM, 2002.

[JMN13] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A. Navas. *Boosting Concolic Testing via Interpolation*. Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), page 48. ACM, 2013.

[JMNS12] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. *TRACER: A Symbolic Execution Tool for Verification*. Proceedings of the 24th International Conference on Computer Aided Verification (CAV), volume 7358 of *LNCS*, pages 758–766. Springer-Verlag, 2012.

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. *The CLP(R) Language and System*. ACM Transactions on Programming Languages and Systems (TOPLAS), volume 14, number 3, pages 339–395. 1992.

[JNS12] Joxan Jaffar, Jorge A. Navas, and Andrew E. Santosa. *Unbounded Symbolic Execution for Program Verification*. Proceedings of the 2nd International Conference on Runtime Verification (RV), volume 7186 of *LNCS*, pages 396–411. Springer-Verlag, 2012.

[JSV09] Joxan Jaffar, Andrew Santosa, and Razvan Voicu. *An Interpolation Method for CLP Traversal*. Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP), volume 5732 of *LNCS*, pages 454–469. Springer-Verlag, 2009.

[Kin76] James C. King. *Symbolic Execution and Program Testing*. Communications of the ACM, volume 19, number 7, pages 385–394. 1976.

[Lem15] Thomas Lemberger. *Symbolic Execution in CPAchecker*. Technical report, Chair of Software Systems, University of Passau, 2015.

[McM10] Kenneth L. McMillan. *Lazy Annotation for Program Testing and Verification*. Proceedings of the 22nd International Conference on Computer Aided Verification (CAV), volume 6174 of *LNCS*, pages 104–118. Springer-Verlag, 2010.

[MS07] Rupak Majumdar and Koushik Sen. *Hybrid Concolic Testing*. Proceedings of the 29th International Conference on Software Engineering (ICSE), pages 416–425. IEEE Computer Society, 2007.

[MSB11] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, 3rd edition, 2011.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. *Cute: A Concolic Unit Testing Engine for C*. Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), page 263. ACM, 2005.

[SST13] Jiri Slaby, Jan Strejcek, and Marek Trtik. *Compact Symbolic Execution*. Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA), volume 8172 of *LNCS*, pages 193–207. Springer-Verlag, 2013.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, den 9. Juli 2015

Thomas Lemberger