# Effective Symbolic Execution in CPAChecker with compositional CEGAR

June 19, 2015

## Contents

**Abstract**

# 1 Introduction

Software systems are prone to error due to multiple factors: The developers skills, humans' limited understanding of software principles, communication problems in development, missing or sparse documentation and unforeseen interdependencies 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. *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 errorless execution of a program with arbitrary input. Additionally, human testing is a technique only shortly touched in most software engineering educations, resembling art more than science.[14]

Along with these points, the rise of ubiquitious computing and reactive systems complicates the process of reliable verification by testing even more. Always running systems that gather, evaluate and adapt their behaviour to information of their environment pose an immense amount of possible inputs and interactions due to these. This makes it almost impossible to predict the behaviour of such systems through testing.

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 [8] is such a program that yielded excellent performance in the last iterations of the Competition on Software Verification (SV-COMP) [3] [4] [5]. CPAChecker is a framework for *Configurable Software Verification* [7] 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 her simplicity, it can't handle complex program characteristics like pointers or non-deterministic values. The predicate CPA, in constrast, is very expressive, but has low efficiency since SAT checks are necessary for computing the feasibility of a program path. The symbolic execution CPA poses something in between those two, not being able to handle some complex program char-

```
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;
  }
}
```

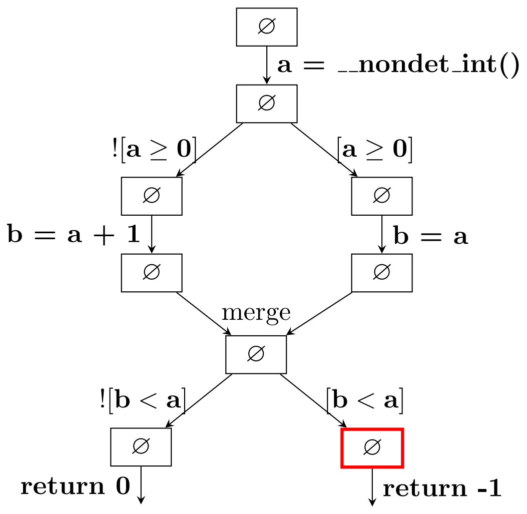
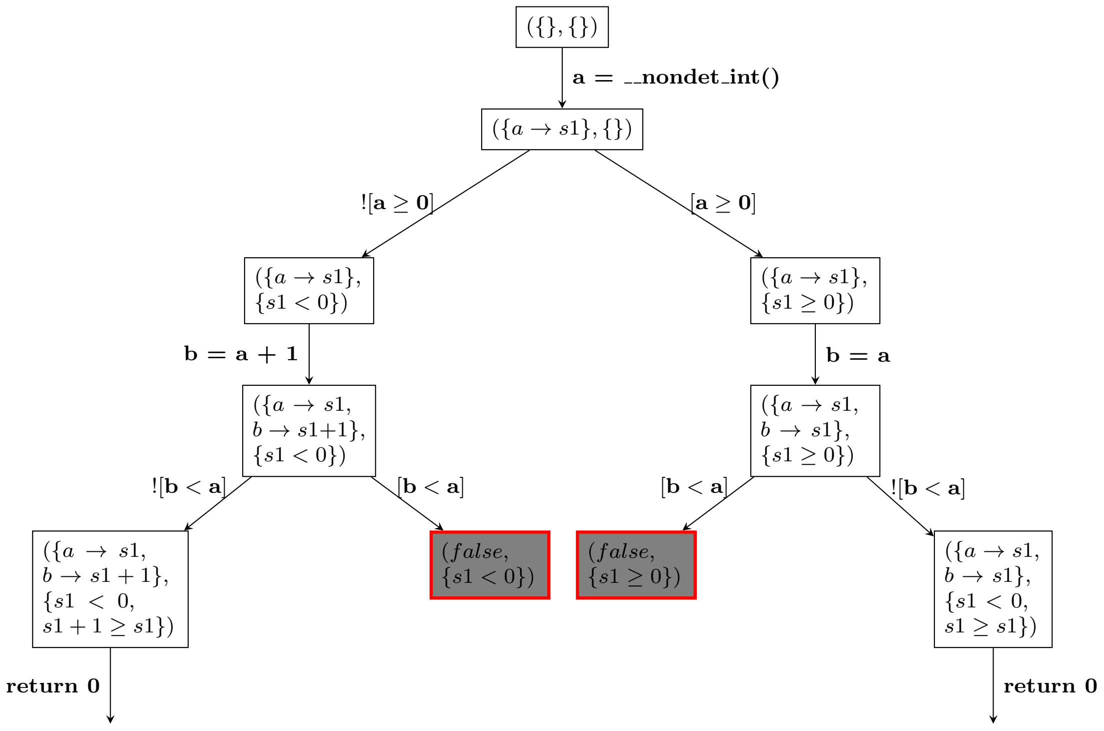


Figure 1: Simple program and its execution by the value analysis CPA.

acteristics 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 displays an example program that uses non-deterministic values and its analysis using the classic value analysis CPA. 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 constrast to this, the symbolic execution CPA is able to handle the non-deterministic assignment to **a** a and its later usage. It returns that the program is safe, correctly. Figure 2 shows this analysis.

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 [13]. 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 occuring assumptions. If a infinite loop occurs, the state space is infinite, too. This problem is called *path explosion.*[1] 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



$(\{\}, \{\})$
a = __nondet_int()
$(\{a \to s1\}, \{\})$
![a ≥ 0]
[a ≥ 0]
$(\{a \to s1\}, \{s1 < 0\})$
$(\{a \to s1\}, \{s1 \geq 0\})$
b = a + 1
b = a
$(\{a \to s1, b \to s1{+}1\}, \{s1 < 0\})$
$(\{a \to s1, b \to s1\}, \{s1 \geq 0\})$
![b < a]
[b < a]
[b < a]
![b < a]
$(\{a \to s1, b \to s1 + 1\}, \{s1 < 0, s1 + 1 \geq s1\})$
$(false, \{s1 < 0\})$
$(false, \{s1 \geq 0\})$
$(\{a \to s1, b \to s1\}, \{s1 < 0, s1 \geq s1\})$
return 0
return 0

Figure 2: Analysis of the program in Listing **??** by the symbolic execution CPA.

values occuring in all encountered assumptions on the currently considered program path. Evaluation in [13] 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 increasing the performance of the symbolic execution CPA. Our main contribution is, along with variations to the existing merge and less-or-equal operators of the CPA, the application of CEGAR [12] to the composition of the two strongly intertwined CPAs value analysis CPA and constraints CPA with precision refinement for both CPAs with two different precision types.

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.

```
 1  extern __VERIFIER_nondet_int();
 2
 3  int main() {
 4    int a = __VERIFIER_nondet_int();
 5    int b;
 6
 7    if (a >= 0) {
 8      b = a;
 9
10    } else {
11      b = a + 1;
12    }
13
14    if (b < a) {
15  ERROR:
16      return -1;
17    }
18  }
```

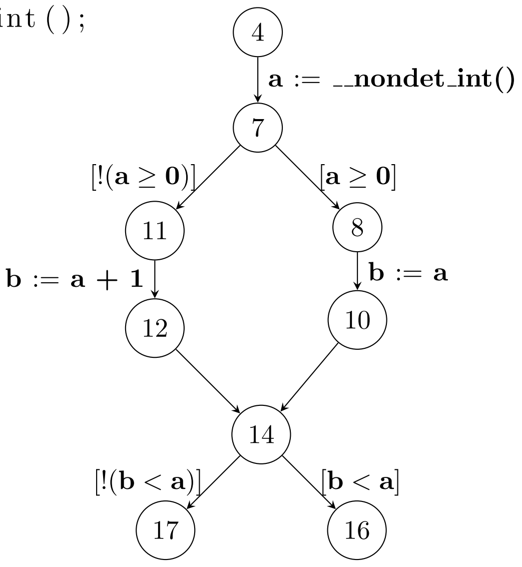


Figure 3: An example CFA.

## 2 Related work

## 3 Theoretical background

### 3.1 General Overview of configurable program analysis

For the sake of simplicity, we focus on a 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. We represent a program by a *control flow automaton* (CFA) [6][11]. 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. A *path* $\sigma$ [10] is a sequence $\langle(op_1, l_1), ..., (op_n, l_n)$ of program locations and their corresponding operations. A path $\sigma$ is a *program path*, if $\sigma$ 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.

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*. For the reachability problem the region of concrete states that violate a given specification is called the *target region* $\sigma^t$.

An abstract domain [7] $D = (C, \mathcal{E}, [\![\cdot]\!])$ 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.*

The basis of automatic software verification is the reachability problem: For a given specification it is derived whether a program state is reachable that violates this specification. Traditionally, two major approaches exist, both working on a reachability tree: 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, which in turn only allows it to handle small programs because of poor performance, program analysis tries to reach high efficiency by looking at only a few chosen characteristics of a program. It does so by using abstract states of a specific, chosen abstract domain to abstract from concrete program states.

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.[7]

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 interpretations, that is the abstract domains to work in and the transfer functions that describe the possible transfers between abstract states,

b) a precision type,

c) a merge operator which controls when two nodes of the reachability tree are merged, i.e. 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* (CPA) [**?**], which is used by the CPA algorithm.

### 3.1.1 Configurable program analysis definition

A CPA with dynamic precision adjustment $\mathbb{D} = (D, \Pi, \rightsquigarrow, \texttt{merge}, \texttt{stop}, \texttt{prec})$ 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:[7][**?**]

   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 $[\![\bot]\!] = \varnothing$,

   b) the join operator has to be precise or over-approximating. That means the join of two abstractes 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 $\forall e, e' \in E : [\![e \sqcup e']\!] \supseteq [\![e]\!] \cup [\![e']\!]$,

   c) $\forall e, e' \in E : e \sqsubseteq e' \Rightarrow [\![e]\!] \subseteq [\![e']\!]$

2. The precision type $\Pi$.

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 $\forall e \in E, g \in G : \bigcup_{e \overset{g}{\rightsquigarrow} e'} [\![e']\!] \supseteq \bigcup_{c \in [\![e]\!]} \{c' | \ c \xrightarrow{g} c'\}$

4. The merge operator $\texttt{merge} : E \times E \times \Pi \to E$, 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 $\texttt{merge}(e, e', \pi)$ can be anywhere between $e'$ and $\top$. Two characteristic merge operators are

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

   and

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

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

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

   and

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

   For $\texttt{stop}^{join}$, 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 $\texttt{prec} : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$. It computes a new abstract state and precision for a given abstract state based on its precision and a set of abstract states with precision. It can both strengthen and weaken an abstract state. The following condition has to hold: $\forall e, \hat{e} \in E, \pi, \hat{\pi} \in \Pi, R \subseteq (E \times \Pi) : (\hat{e}, \hat{p}) = \texttt{prec}(e, \pi, R) \Rightarrow [\![e]\!] \subseteq [\![\hat{e}]\!]$.

The CPA algorithm described in Algorithm 1 uses an arbitrary CPA of this form to solve the reachability problem. Given a CPA $\mathbb{D}$, an initial abstract state $e_0$ to start its computation at and an initial precision $\pi_0$, the algorithm computes the set `reached` of reachable abstract states. As long as the set of abstract states that still have to be processed (`waitlist`) is not empty, an abstract state $e \in \texttt{waitlist}$ is removed from the waitlist and each possible abstract successor $e'$ and its precision $\pi$ is examined:

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

Next, it is checked whether the adjusted state $\hat{e}$ represents any concrete state that violates a property. This is done by $\texttt{isTargetState} : E \to \mathbb{B}$ in Line 8, whose behaviour can be defined arbitrarily. In classic reachability computation, the complete reachable set would be computed first.

Next, each already reached abstract state $e'' \in \texttt{reached}$ is individually merged with the new state $\hat{e}$ with precision $\hat{\pi}$. If the merge weakened $e''$, it

**Algorithm 1** $CPA(\mathbb{D}, e_0, \pi_0)$, adapted from [?]

**Input:** a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \texttt{merge}, \texttt{stop}, \texttt{prec})$, an initial abstract state $e_0 \in E$ with $E$ being the set of elements of $D$, and an initial precision $\pi_0 \in \Pi$.
**Output:** a set of abstract states reachable from $e_0$
**Variables:** `reached` and `waitlist`, both subsets of $E \times \Pi$

```
1:  reached := $\{(e_0, \pi_0)\}$
2:  waitlist := $\{(e_0, \pi_0)\}$
3:  while waitlist $\neq \varnothing$ do
4:      choose $(e, \pi)$ from waitlist
5:      remove $(e, \pi)$ from waitlist
6:      for all $e'$ with $e \rightsquigarrow (e', \pi)$ do
7:          $(\hat{e}, \hat{\pi})$ := prec($e', \pi$, reached)          ▷ Precision adjustment
8:          if isTargetState($\hat{e}$) then
9:              return (reached $\cup \{(\hat{e}, \hat{\pi})\}$, waitlist)
10:         for all $(e'', \pi'') \in$ reached do
11:             $e_{new}$ := 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 $\neg$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
```

and its precision are replaced with the weakened state and the new precision $\hat{\pi}$ in `reached` and `waitlist` (Lines 11 - 14). 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 is continued with the next element in the waitlist. If the waitlist is empty, there are no more reachable states and the reached set is returned.

## 3.2 Basic definition of CPAs used in this paper

A definition of all CPAs important to this paper follows.

### 3.2.1 Composite CPA

It is often useful to combine multiple CPAs to use their individual strengths and mitigate their weaknesses. A composition of two CPAs [?] can be expressed as $(\mathbb{D}_1, \mathbb{D}_2, \Pi_\times, \rightsquigarrow_\times, \texttt{merge}_\times, \texttt{stop}_\times, \texttt{prec}_\times)$. We will extend this definition to allow the composition of an arbitrary number of CPAs. 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 $\texttt{merge}_\times : (E_1 \times E_2) \times (E_1 \times E_2) \times \Pi_\times \to (E_1 \times E_2)$.

5. A composite termination check $\texttt{stop}_\times : (E_1 \times E_2) \times 2^{E_1 \times E_2} \times \Pi_\times \to \mathbb{B}$.

6. A composite precision adjustment

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

The three composite operators 3. - 5. use the corresponding operators of the contained CPAs $\mathbb{D}_1$ to $\mathbb{D}_i$ as well as multiple *strenghtening operators* $\downarrow_j$ and *compare relations* $\preceq_j$. 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 $1 \leq k, l \leq i$ and $k \neq l$ for $i$ being the number of CPAs in the composition. 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 $\mathcal{C} = (D_\times, \Pi_\times, \rightsquigarrow_\times, \texttt{merge}_\times, \texttt{stop}_\times, \texttt{prec}_\times)$ 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

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

. 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 $\texttt{merge}^{sep}$, i.e. no merge is performed.

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

### 3.2.2 Location CPA

The location CPA [6] [?] $\mathbb{L} = (D_{\mathbb{L}}, \widetilde{\Pi}, \rightsquigarrow_{\mathbb{L}}, \texttt{merge}^{sep}, \texttt{stop}^{sep}, \widetilde{\texttt{prec}})$ is a CPA that analyses 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 precisions that only contains a single precision. This means that all abstract states have the same precision all the time.

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 $\texttt{merge}^{sep}$, defined as $\texttt{merge}^{sep}(l, l', \pi) = l'$ for all $l, l' \in \mathcal{L}$.

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

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

### 3.2.3 Predicate CPA

A predicate is a boolean formula using linear-arithmetic expressions and equality with uninterpreted functions. The predicate CPA [6] [?] uses predicate abstraction [2] to compute abstract states from a formula $\phi$ and a set

$\pi$ 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 $\pi$ that is implied by $\phi$. The boolean predicate abstraction $(\phi)^{\pi}_{B}$ is the strongest boolean combination of predicates from $\pi$ that is implied by $\phi$. In this work, we will only look at cartesian predicate abstraction because of its greater simplicity. For a set $r \subseteq \pi$, $\varphi_r$ denotes the conjunction of all predicates in $r$, with $\varphi_{\{\}} = \mathit{true}$.

The predicate CPA $\mathbb{P} = (D_{\mathbb{P}}, \Pi_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \texttt{merge}^{sep}, \texttt{stop}^{sep}, \widetilde{\texttt{prec}})$ 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 ($\mathit{true}$) and represents all possible concrete states. The bottom element $\bot = \{\mathit{false}\}$ represents no concrete state. A state $r$ is less or equal another state $r'$, if $r$ contains all predicates of $r'$, formally $r \sqsubseteq r'$ if $r \supseteq r'$. The join of two 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 precision type $\Pi_{\mathbb{P}} = 2^P$ describes the precision of an abstract state as a set of predicates. If predicate $p \in P$ is in a precision $\pi$, $p$ is tracked by the analysis when $\pi$ 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 $\pi$ 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 $\varphi$ 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 $\texttt{merge}^{sep}$, defined as $\texttt{merge}^{sep}(r, r', \pi) = r'$ for all $r, r' \in \mathcal{P}$.

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

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

### 3.2.4 Value analysis CPA

The value analysis CPA [9] tracks integer values for all program variables explicitly. It does so by using *abstract variable assignments* [9]. An abstract variable assignment $v : X \nrightarrow \mathbb{Z} \cup \{\bot\}$ is a partial function that maps program variables $x \in X$ to integer values - if their assignment is known - or to $\bot$, if no possible value assignment exists. An abstract variable assignment $v$ is *contradicting* if $v(x) = \bot$ 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 $x \in \texttt{def}(v) \cap \texttt{def}(v') : v(x) = v'(x)$. The *conjunction* $v \wedge v'$ is defined as

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

We define the *definition range* of a partial function $f$ as $\texttt{def}(f) = \{x | \exists y : (x, y) \in f\}$ 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 $\mathbb{C} = (D_{\mathbb{C}}, \Pi_{\mathbb{C}}, \rightsquigarrow_{\mathbb{C}}, \texttt{merge}^{sep}, \texttt{stop}^{sep}, \widetilde{\texttt{prec}})$ 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 $\texttt{def}(v') \subseteq \texttt{def}(v)$ and for all $x \in \texttt{def}(v') : v(x) = v'(x)$ or $v(x) = \bot_{\mathcal{Z}}$. This means that a state $v$ is less or equal a 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, $[\![v]\!] = \{c \in C | \ c(x) = v(x) \text{ for all } x \in \texttt{def}(v)\}$. If an abstract state $v$ contains an impossible variable assignment, that is $v(x) = \bot_{\mathcal{Z}}$ for any $x \in \texttt{def}(v)$, then it represents no concrete state: $[\![v]\!] = \varnothing$.

2. The set of precisions $\Pi_{\mathbb{C}} = L \rightarrow 2^X$. 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 $x \in \texttt{def}(v')$:

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

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

   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

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

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

4. The merge operator $\texttt{merge}^{sep}$. That means that no merge is performed. This is the only aspect in which constant propagation CPA [6] and value analysis CPA differ.

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

6. The precision adjustment $\widetilde{\texttt{prec}}$ that does not change anything: $\widetilde{\texttt{prec}}(v, \pi, R) = (v, \pi)$. 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 [13] without dynamic precision adjustment) 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 $\mathbb{C}_{\mathbb{S}} = (D_{\mathbb{C}_{\mathbb{S}}}, \Pi_{\mathbb{C}_{\mathbb{S}}}, \rightsquigarrow_{\mathbb{C}_{\mathbb{S}}}, \texttt{merge}_{\mathbb{C}_{\mathbb{S}}}, \texttt{stop}^{sep}, \widetilde{\texttt{prec}})$ consists of:

1. The abstract domain $D_{\mathbb{C}_{\mathbb{S}}} = (C, \mathcal{E}, [\![\cdot]\!])$ with the set $C$ of concrete states, the semi-lattice $\mathcal{E}$ and the concretization function $[\![\cdot]\!]$. The semi-lattice $\mathcal{E} = (V_{\mathbb{C}_{\mathbb{S}}}, \top_{\mathbb{C}_{\mathbb{S}}}, \bot, \sqsubseteq, \sqcup)$ is defined by the set of abstract symbolic value assignments $V_{\mathbb{C}_{\mathbb{S}}} = X \nrightarrow \mathcal{Z}_{\mathbb{C}_{\mathbb{S}}}$ mapping program variables in its definition range to values of $\mathcal{Z}_{\mathbb{C}_{\mathbb{S}}} = \mathbb{Z} \cup S \cup \{\bot_{\mathcal{Z}}\}$. The value range of an abstract variable assignment of the type $V_{\mathbb{C}_{\mathbb{S}}}$ 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 $V_{\mathbb{C}_{\mathbb{S}}}$ 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 $\top_{\mathbb{C}_{\mathbb{S}}} = \varnothing$ represents no known assignment.

   For two abstract states $v, v' \in V_{\mathbb{C}_{\mathbb{S}}}$, $v \sqsubseteq v'$, if all of the following conditions hold: (a) $v'$ must only contain value assignments of $v$, that is $\texttt{def}(v') \subseteq \texttt{def}(v)$, (b) for every concrete or invalid assignment of $v$, $v'$ must contain the same or a weaker one, that means the same value or a concrete value for $\bot_{\mathcal{Z}}$, formally put

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

   (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 $\forall x \in \texttt{def}(v') : v'(x) \in S \Rightarrow v(x)$ results from $v'(x)$ by replacing all $i \in S_I$ occurring in $v'(x)$ with $\texttt{alias}(i)$. It can be thought of simpler or more precise less-or-equal operators. Such operators will be examined later. Here, we use this operator (with some fixes to its origin) simply because it was used in our previous work [13].

   Note that with this operator $v' \sqsubseteq v \Rightarrow [\![v']\!] \subseteq [\![v]\!]$, but in general $[\![v']\!] \subseteq [\![v]\!] \not\Rightarrow v' \sqsubseteq v$.

   The join $\sqcup : V_{\mathbb{C}_{\mathbb{S}}} \times V_{\mathbb{C}_{\mathbb{S}}}$ 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 precision type $\Pi_{\mathbb{C}_{\mathbb{S}}} = L \to 2^X$. 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 $\rightsquigarrow_{\mathbb{C}_\mathbb{S}}$ contains the transfer $v \overset{g}{\rightsquigarrow}_{\mathbb{C}_\mathbb{S}} v''$, if one of the following conditions is true:

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

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

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

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

   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

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

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

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

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

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

6. The precision adjustment $\widetilde{\texttt{prec}}$ that does not change anything: $\widetilde{\texttt{prec}}(v, \pi, R) = (v, \pi)$. 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 [13] without dynamic precision adjustment) 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 $\mathbb{A} = (D_{\mathbb{A}}, \Pi_{\mathbb{A}}, \rightsquigarrow_{\mathbb{A}}$ $, \texttt{merge}_{\mathbb{A}}, \texttt{merge}^{sep}, \widetilde{\texttt{prec}})$ 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 $\mathcal{A} = (2^{\gamma}, \top_{\mathbb{A}}, \bot, \sqsubseteq, \sqcup)$ are subsets of the set $\gamma$ of all possible boolean expressions over the possible values of the symbolic value analysis CPA, $\mathcal{Z}_{\mathbb{C}_{\mathbb{S}}}$, 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 $false$. $\sqsubseteq$ is defined in the following way: For two given states $a, a' \subseteq \gamma$, $a$ is less or equal $a'$, that is $a \sqsubseteq a'$, if a bijective function $\texttt{alias} : S_I \to S_I$ exists, so that $a'' \subseteq a$ with $a''$ resulting from $a'$ by replacing all symbolic identifiers $i \in S_I$ occurring in the constraints of $a'$ with $\texttt{alias}(i)$. 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:

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

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

2. The set $\Pi_{\mathbb{A}} = L \to 2^{\gamma^+}$ 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 $\mathcal{Z}_{\mathbb{C}_{\mathbb{S}}}$. The constraints of $\gamma^+$ do not contain any program variables, but the only variables occuring in them are symbolic identifiers of $S_I$.

   A second option is to use the precision $\Pi_{\mathbb{C}_{\mathbb{S}}} = L \to 2^X$ of the symbolic value analysis CPA. A constraint $p$ is tracked by $\pi \in \Pi_{\mathbb{C}_{\mathbb{S}}}$ at a location $l$, if $\pi(l)$ contains all program variables $p$ originated from.

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

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. Or,

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

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

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

6. The precision adjustment $\widetilde{\texttt{prec}}$ that does not change anything: $\widetilde{\texttt{prec}}(r, \pi, R) = (r, \pi)$. 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: Composition of Location CPA, Symbolic Value Analysis CPA and Constraints CPA

The symbolic execution CPA [13] 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 $(\mathbb{L}, \mathbb{C}_{\mathbb{S}}, \mathbb{A}, \Pi_{\mathbb{S}}, \rightsquigarrow_{\mathbb{S}}, \texttt{merge}_{\mathbb{S}}, \texttt{stop}_{\mathbb{S}}, \texttt{prec}_{\mathbb{S}})$. It consists of:

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

2. The precision type $\Pi_{\mathbb{S}} = \Pi_{\mathbb{C}_{\mathbb{S}}} \times \Pi_{\mathbb{A}}$ 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 $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} (l', \widetilde{\pi})$, $v \overset{g}{\rightsquigarrow}_{\mathbb{C}_{\mathbb{S}}} (v', \pi_{\mathbb{C}_{\mathbb{S}}})$, if:

a) $g = (\cdot, \texttt{assume}(p), \cdot)$ and $\downarrow_{\mathbb{A},\mathbb{C}_{\mathbb{S}}} (a'', v') = a'$ is defined, with $a \overset{g}{\rightsquigarrow}_{\mathbb{A}} (a'', \pi_{\mathbb{A}})$, or

b) $g = (\cdot, w := e, \cdot)$ and $a \overset{g}{\rightsquigarrow} (a, \pi_{\mathbb{A}})$,

and if $\pi = (\widetilde{\pi}, \pi_{\mathbb{C}_{\mathbb{S}}}, \pi_{\mathbb{A}})$.

It uses the strengthening operator $\downarrow_{\mathbb{A},\mathbb{C}_{\mathbb{S}}}: 2^{\gamma} \times V_{\mathbb{C}_{\mathbb{S}}} \to \gamma^{+}$ that strengthens an abstract state of the constraints CPA by using an abstract state of the symbolic value analysis CPA. $\downarrow_{\mathbb{A},\mathbb{C}_{\mathbb{S}}} (a'', v) = a'$ 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 $\bot_{\mathbb{Z}}$. We define a constraint containing $\bot_{\mathbb{Z}}$ as *false*, though, and as such, the strengthen operator is not defined if $\bot_{\mathbb{Z}}$ is occurs.

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

4. The merge operator $\texttt{merge}_{\mathbb{S}} = \texttt{merge}^{agree}$ 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 $\texttt{stop}_{\mathbb{S}}$ 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 $\texttt{prec}_{\mathbb{S}}$ 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.

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

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

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

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 precision type of the constraints CPA: If $\Pi_{\mathbb{A}}$ is used, which stores all tracked constraints explicitely, the adjustment is defined as

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

If $\Pi_{\mathbb{C}_{\mathbb{S}}}$ 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 $\pi$ being the current precision.

## 3.3 Basic CEGAR and its algorithm

### 3.3.1 CEGAR and interpolation in general

Counterexample-guided abstraction refinement (CEGAR) [12] 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,"[12] 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.[9] 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, [12] good heuristics have to be used to find good refinements. Interpolation [**?**] is one such technique in a boolean context that is used for refinement of both the predicate CPA and value analysis CPA.

### 3.3.2 CEGAR and interpolation in the context of configurable software verification

To apply CEGAR and interpolation to configurable software verification, a simple modification has to be made to the CPA algorithm. Instead of passing it an initial state $e_0$ and an initial precision $\pi_0$, we use an initial reached set $R_0$ and initial waitlist $W_0$ (Alg. 2). This way we can control at which point the analysis continues after a refinement was performed.

**Algorithm 2** $CPA(\mathbb{D}, R_0, W_0)$, adapted from [9]

**Input:** a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathtt{merge}, \mathtt{stop}, \mathtt{prec})$, a set $R_0 \subseteq (E \times \Pi)$ of initial 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 abstract states reachable from $R_0$ with their precision and a subset of frontier abstract states with their 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                ▷ from here on the same as before
4:     ...
```

Now that the CPA algorithm is able to use precisions created in a refinement procedure, we use it as a part of our complete CEGAR algorithm. Algorithm 3 uses a CPA using dynamic precision adjustment $\mathbb{D}$, an initial state $e_0$ and an initial precision $\pi_0$ 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, 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` : $\Sigma \to \Pi$ with $\Sigma$ 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 precision type $\Pi$ 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 $\psi$ is a Craig interpolant [**?**] for two boolean formulas $\gamma^-$ (called prefix) and $\gamma^+$ (called suffix), if the following three conditions are fulfilled:

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

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

**Algorithm 3** $CEGAR(\mathbb{D}, e_0, \pi_0)$, adapted from [9]

```
Input: a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \texttt{merge}, \texttt{stop}, \texttt{prec})$ 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 $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$
 1:
 2: reached := $\{(e_0, \pi_0)\}$
 3: waitlist := $\{e_0, \pi_0\}$
 4: $\pi := \pi_0$
 5: while true do
 6:     (reached, waitlist) := $CPA(\mathbb{D}$, reached, waitlist)
 7:     if waitlist = $\varnothing$ then return $safe$
 8:     else
 9:         $\sigma$ := extractErrorPath(reached)
10:         if isFeasible($\sigma$) then            ▷ error path feasible: report bug
11:             return $unsafe$
12:         else▷ error path infeasible: refine and restart from the beginning
13:             $\pi := \pi \cup$ refine($\sigma$)
14:             reached := $(e_0, \pi)$
15:             waitlist := $(e_0, \pi)$
```

c) $\psi$ 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 [9] as well as in the theory of linear arithmetics [**?**].

Our work is strongly based on the refinement technique for abstract variable assignments. The strongest-post operator $\text{SP}_{op}$ 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 $\gamma_\sigma = \langle op_1, ..., op_n \rangle$: $\text{SP}_{\gamma_\sigma}(v) = \text{SP}_{op_n}(\text{SP}_{op_{n-1}}(... \text{SP}_{op_1}(v)...))$. We use strongest-post operators during interpolation and refinement to evaluate paths.

The strongest-post operator $\text{SP}_{op}$ is defined in the following way: For an assignment operation $s := exp$, $\text{SP}_{s:=exp}(v) = v_{|X \setminus \{s\}} \wedge v_{s:=exp}$ 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

**Algorithm 4** `interpolate`$(\gamma^-, \gamma^+)$, adapted from [9]

```
Input: two constraint sequences $\gamma^-$ and $\gamma^+$, with $\gamma^- \wedge \gamma^+$ contradicting
Output: a constraint sequence $\Gamma$, which is an interpolant for $\gamma^-$ and $\gamma^+$
Variables: an abstract variable assignment $v$
1: $v := \mathrm{SP}_{\gamma^-}(\varnothing)$
2: for all $x \in \texttt{def}(v)$ do
3:     if $\mathrm{SP}_{\gamma^+}(v_{|\texttt{def}(v)\setminus\{x\}})$ is contradicting then
4:         $v := v_{|\texttt{def}(v)\setminus\{x\}}$ ▷ $x$ not relevant, should not occur in interpolant
5: $\Gamma := \langle\rangle$
6: for all $x \in \texttt{def}(v)$ do
7:     $\Gamma := \Gamma \wedge \langle \texttt{assume}(x = v(x))\rangle$
8: return $\Gamma$
```

**Algorithm 5** `refine`$(\sigma)$, adapted from [10]

```
Input: infeasible error path $\sigma = \langle(op_1, l_1), ..., (op_n, l_n)\rangle$
Output: precision $\pi$
Variables: interpolating constraint sequence $\Gamma$
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$(\Gamma)$
7: return $\pi$
```

operation `assume`$(p)$, $\mathrm{SP}_{\texttt{assume}(p)}(v) = v'$ 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.

The algorithm for interpolation in the domain of abstract variable assignments is shown in Algorithm 4. For a prefix $\gamma^-$ and a suffix $\gamma^-$, the abstract variable assignment $v$, that results from applying $\gamma^-$ 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 $\gamma^+$ is contradicting. If it is not, it can be removed from $v$. After all variable assignments are checked, $v$ only contains variable assignments that are necessary to prove that $\gamma^+$ is contradicting. From these, the interpolant is built (Lines 6 - 8).

The interpolants produced are used in the refinement of the precision

(Alg. 5). 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. This approach realizes the lazy abstraction technique [**?**]. The algorithm starts with an initial, empty interpolant $\Gamma$ and empty precision $\pi$ with $\pi(l) = \varnothing$ for all $l \in L$. For each location $(l_i, op_i)$ on the error path, the suffix $\gamma^+$ 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). 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:

$$\texttt{extractPrecision}(\Gamma) = \{x|\ (x, z) \in \mathrm{SP}_\Gamma(\varnothing) \text{ and } z \neq \bot_\mathcal{Z}\}.$$

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. 3) 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.

Refinement in the domain of linear arithmetics, 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 arithmetics and abstract variable assignments, which we will use when applying CEGAR to the symbolic execution CPA.

# 4 Definitions of newly introduced concepts

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 performace 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 A different 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 transfering 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 [?]). 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

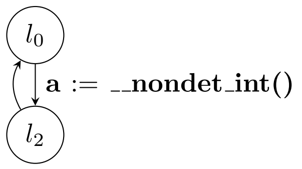
$$\texttt{merge}(a, a', \pi) = \begin{cases} a' \setminus \neg Q & \text{if } a \sqsubseteq a' \setminus \neg Q \\ a' & \text{otherwise} \end{cases}$$

with $\neg Q = \{\neg p |\ p \in a \wedge \neg p \in a'\}$ and $Q = \{p |\ p \in a \wedge \neg p \in a'\}$. 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 $a \sqsubseteq a \setminus Q$.

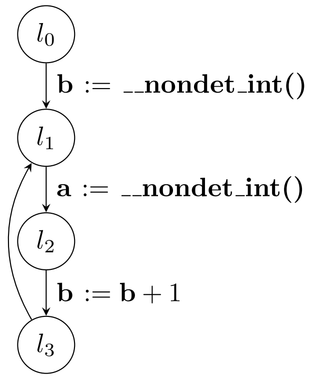
This condition is automatically checked by the $\texttt{merge}^{agree}$ operator, so we can simply use $\texttt{merge}(a, a', \pi) = a' \setminus \neg Q$.

## 4.2 Different less-or-equal operators

The less-or-equal operator is the operator executed the most often during analyses as $\texttt{stop}^{sep}$ 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. Because of this, its speed and precision can make a great difference for the performance of our analysis.



(a) An example CFA


$l_0$
$\mathbf{b} := \_\_\mathbf{nondet\_int()}$
$l_1$
$\mathbf{a} := \_\_\mathbf{nondet\_int()}$
$l_2$
$\mathbf{b} := \mathbf{b} + 1$
$l_3$

(b) An example CFA

Figure 4: Two CFAs with non-deterministic assignments in a loop

The less-or-equal operators we used for symbolic value analysis CPA and constraints CPA using an `alias` function try to be more precise than a simple subset check. Aliasing proves only useful, though, when a non-deterministical value is assigned in a loop. A simple example can be seen in Figure 4a. After the first iteration of the loop, an abstract variable assignment $v = \{a \to s1\}$ exists in the reached set, with $s1 \in S_I$. At the second iteration of the loop, the new abstract variable assignment $v' = \{a \to s2\}$ with $s2 \in S_I$ is created, but not added to the reached set, since $\texttt{stop}^{sep}(v', \{v\}, \pi) = \mathit{true}$ since $v' \sqsubseteq v$ using the aliasing less-or-equal operator.

This is only useful as long as no variables are both assigned values and accessed inside the loop, as the next example shows. In Figure 4b a second program variable, **b**, gets assigned a non-deterministic value outside the loop, which is used inside the loop to compute a new value for **b**. After the first iteration of the loop, an abstract variable assignment $v = \{a \to s1, b \to s2 + 1\}$ is part of the reached set. At the second iteration, a second abstract variable assignment $v' = \{a \to s3, b \to s2 + 2\}$ is created and no fitting alias can be found because of $s2 + 1$ not fitting $s2 + 2$. A new abstract variable assignment is added like this at every new loop. Because of this strong restriction, we assume that the aliasing proves useful in only few cases.

Additionally, the computation of this `alias` function increases the complexity of the operator to a high degree (depending on data-structures and the used algorithm). It is worth evaluating a simple subset check as less-or-equal operator to see whether the aliasing's higher precision in some special cases actually justifies its lower performance. The simple less-or-equal operator performing a subset check for the symbolic value analysis CPA is defined as $v \sqsubseteq_{sub} v'$ if $v \supseteq v'$ and for the constraints CPA it is defined as $a \sqsubseteq_{sub} a'$ if $a \supseteq a'$.

Since a constraints CPA's abstract state $a$ is interpreted as the conjunction of its constraints $\varphi_a$, 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 $\varphi_a$ implies a formula $\varphi_{a'}$ and $c$ satisfies $\varphi_a$, 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 $\sqsubseteq_{sub}$ and a higher one than the aliasing less-or-equal operator as long as no aliasing is necessary/possible, but requires SAT checks, which are definitely worse in performance than merely checking whether one set is the subset of another. If aliasing is possible for proving a less-or-equal relation, though, we can't compare implication to the aliasing less-or-equal operator directly. Implication handles symbolic identifiers as variables and as such will not be able to prove $a \Rightarrow a'$ if $a'$ contains any variable not contained in $a$.

## 4.3 Location/Frequency of SAT checks

### 4.3.1 After every assume

### 4.3.2 At every target location only

## 4.4 CEGAR for Symbolic Value Analysis + ConstraintsCPA

For using the symbolic execution CPA with CEGAR, we have to define a refinement procedure that returns a precision of type $\Pi_{\mathbb{S}}$ 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 precision type that fits $\Pi_{\mathbb{S}}$. 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 arithmetics.

### 4.4.1 Refinement based on refinement for abstract variable assignments

We adjust the refinement algorithm for abstract variable assignments (Alg. 5). The feasibility check of an error path $\sigma$ is performed by executing the symbolic execution CPA with full precision for all program locations $l$. If the error location of $\sigma$ 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 $\mathrm{SP}^{\mathbb{S}}_{op} : V_{\mathbb{C}_{\mathbb{S}}} \times 2^{\gamma^+} \rightarrow V_{\mathbb{C}_{\mathbb{S}}} \times 2^{\gamma^+}$. It is the composition of the transfer relations of the symbolic value analysis CPA and the constraints CPA, as well as the strengthen operator

**Algorithm 6** `interpolate`$_{\mathbb{S}}(\gamma^-, \gamma^+)$, a modified version of Alg. 4

```
Input: two constraint sequences $\gamma^-$ and $\gamma^+$, with $\gamma^- \wedge \gamma^+$ contradicting
Output: a constraint sequence $\Gamma$, which is an interpolant for $\gamma^-$ and $\gamma^+$
Variables: an abstract variable assignment $v$ and a constraints state $a$
 1: $(v, a) := \mathrm{SP}^{\mathbb{S}}_{\gamma^-}(\varnothing)$
 2: for all $p \in a$ do
 3:     if $\mathrm{SP}^{\mathbb{S}}_{\gamma^+}(v, a \setminus \{p\})$ is contradicting then
 4:         $a := a \setminus \{p\}$        $\triangleright$ $p$ not relevant, should not occur in interpolant
 5: for all $x \in \mathtt{def}(v)$ do
 6:     if $\mathrm{SP}^{\mathbb{S}}_{\gamma^+}(v_{|\mathtt{def}(v)\setminus\{x\}}, a)$ is contradicting then
 7:         $v := v_{|\mathtt{def}(v)\setminus\{x\}}$ $\triangleright$ $x$ not relevant, should not occur in interpolant
 8: $\Gamma := \langle\rangle$
 9: for all $p \in a$ do
10:     $\Gamma := \Gamma \wedge \langle \mathtt{assume}(p) \rangle$
11: for all $x \in \mathtt{def}(v)$ do
12:     $\Gamma := \Gamma \wedge \langle x := v(x) \rangle$
13: return $\Gamma$
```

$\downarrow_{\mathbb{A},\mathbb{C}_{\mathbb{S}}}$ to create useful constraints states. The result of $\mathrm{SP}^{\mathbb{S}}_{op}$ is contradicting if $\downarrow_{\mathbb{A},\mathbb{C}_{\mathbb{S}}}$ 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:

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

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 $\rightsquigarrow_{\mathbb{C}_{\mathbb{S}}}$ 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 $\mathrm{SP}^{\mathbb{S}}_{op}$ unambigious. 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. 6) allows the computation of an interpolant for a prefix $\gamma^-$ and a suffix $\gamma^+$ at a specific program location based on the semantics of the symbolic execution CPA. Since we want to create an interpolant $\Gamma$ that contains all information necessary for proving that $\mathrm{SP}^{\mathbb{S}}_{\Gamma\wedge\gamma^+}$ is contradicting, not only abstract variable assignments but also constraints have to be considered. First, we compute

**Algorithm 7** $\texttt{refine}_{\mathbb{S}}(\sigma)$, a modified version of Alg. 5.

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

the strongest-post condition $(v, a)$ for the prefix $\gamma^-$. Similar to interpolation for abstract variable assignments, we then eliminate all constraints from $a$ that are not necessary for proving that $\gamma^+$ 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 6, we build the interpolant $\Gamma$ 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 distinguish between both domains. It is not wrong to only use `assume` operations, but it would unnecessarily increase the precision for the constraints CPAand 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. Additionaly, it is also possible to eliminate variable assignments first, and constraints second. We will examine all three possibilities later in detail.

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 $\Gamma$, we define the `extractPrecision` function as the composition of two new functions, each of which extracts the precision for one of these CPAs based on $\Gamma$:

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

with

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

and

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

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 7.

### 4.4.2 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 $\Pi_{\mathbb{S}}$ 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 $\sigma$ 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 precision type: 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 precision type $\Pi_{\mathbb{A}} = L \to 2^{\gamma^+}$ as part of $\Pi_{\mathbb{S}}$, but only $\Pi_{\mathbb{C}_{\mathbb{S}}}$ (see Section 3.2.6). Secondly, the precision type $\Pi_{\mathbb{P}} = 2^P$ 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 then $\texttt{refine}_{\mathbb{S}}$. 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 8 shows the refinement procedure delegating to predicate CPA's refinement. After computing the precision $\pi'$ of the predicate CPA, the variables used in the predicates of $\pi'$ are extracted and assigned as the precision $\pi_{\mathbb{C}_{\mathbb{S}}}(l)$ for

**Algorithm 8** $\texttt{refine}'_{\mathbb{S}}(\sigma)$

```
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 $\pi' \in \Pi_\mathbb{P} = 2^P$
1: $(\pi_{\mathbb{C}_\mathbb{S}}(l), \pi_\mathbb{A}(l)) := (\varnothing, \varnothing)$ for all program locations $l$
2: $\pi' = \texttt{refine}_\mathbb{P}(\gamma)$
3: $\pi_{\mathbb{C}_\mathbb{S}}(l) := \texttt{extractPrecision}'_\mathbb{S}(\pi')$ for all $l$
4: return $(\pi_{\mathbb{C}_\mathbb{S}}, \pi_{\mathbb{C}_\mathbb{S}})$
```

every location $l$. The precision $\pi_{\mathbb{C}_\mathbb{S}}$ is then returned as precision for both symbolic value analysis CPA and constraints CPA.

Figure 5 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 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.

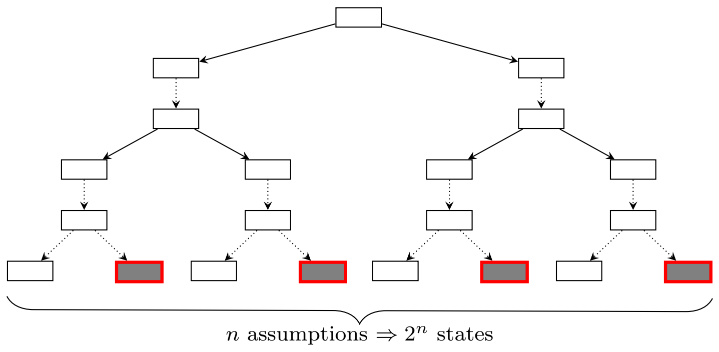
# 5 Implementations of used CPAs in CPAchecker

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

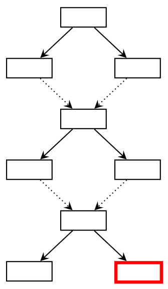
## 5.1 Basic implementation

We implemented our algorithm in the framework for configurable program verification CPAChecker[8]. 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.

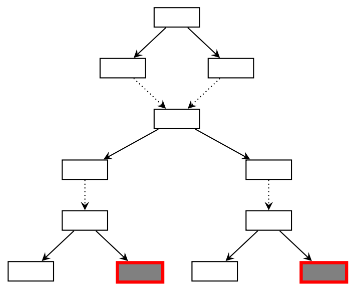
A wide array of CPA implementations already exists, which we can utilize. The CPA interface equals the theoretical definition, but is extended with an initial state and initial precision, which are used as initial parameters of the CPA algorithm. The interface of the transfer relation is extended to include strengthening for the designated CPA, as almost always a com-



(a) Analysis without CEGAR



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



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

Figure 5: Symbolic execution analysis of a program with and without CEGAR. The red rectangles are abstract states at an infeasible target location. If they are filled grey, the analysis sees them as infeasible.

posite 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 $\texttt{merge}^{agree}$, 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 $\Pi_{\mathbb{C}_\mathbb{S}}$. As such we implemented the two auxiliary precision functions $\text{prec}_{\mathbb{C}_\mathbb{S}}$ 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 [13]. 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 $\gamma^+$.

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 precision type $\Pi_{\mathbb{C}_\mathbb{S}}$ for the constraints CPA, see Section 3.2.6 and 4.4.1), 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 $\gamma^+$ 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` and `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 analysing

```
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.

**Algorithm 9** GetDefiniteAssignments($F, M$)

```
Input: A boolean formula $F$ and a model $M = S_I \to \mathbb{Z}$ that satisfies $F$
Output: A map $D \subseteq M$ of definite assignments
  for all $(s, n) \in M$ do
      if $F \wedge s \neq n$ is unsatisfiable then
          $D := D \cup (s, n)$
  return $D$
```

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. 9)

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 satisifiable 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 and less error-prone code, this is changed in our current implementation (see Section 5.3).

As a third optimization, we periodically delete all constraints that do not hold any information, directly or indirectly, about a variable with symbolic value. This is the case for a constraint if none of the symbolic identifiers it contains are present in the current value analysis state and if either none of them occur in another constraint that holds information about a program variable with symbolic value, or if at least one of them only occurs in this one constraint and does not have a definite value. By doing this, we reduce the number of constraints to the minimum without losing any information and accelerate the SAT checks at every assumption edge due to fewer variables in boolean formulas.

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 lots of redundant object creations.

We perform strenghtening 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

As already mentioned, we do not remove constraints that only consist of symbolic identifiers with definite assignments, anymore. Instead, we update stored boolean formulas after computing new definite assignments by replacing all formulas that contain a variable with a new definite assignment with a new version using this definite assignment. This has two advantages: First, we do not have to consider definite assignments when examining constraints states, because all information concerning symbolic identifiers is present in a state's constraints. Second, we still get the performance benefit of minimizing the amount of symbolic identifiers we wanted to achieve by deleting trivial constraints, as the formulas representing the constraints use the definite assignments. As a consequence, a boolean formula representing a trivial constraint does not contain any variable and its satisfiability can be checked easily. Figure 6 illustrates this procedure using some example formulas. After checking that the set of formulas (in the figure called "old formulas") is satisfiable, new definite assignments are computed. For $a > 0$,

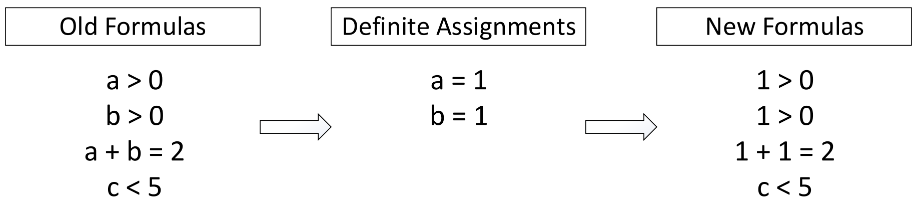


Figure 6: Illustration of the simplification of a constraints state's formulas by replacing variables with definite assignments

$b > 0$ and $a + b = 2$, $a$ and $b$ both have to be 1 to fullfil the conjunction of these formulas. Variable $c$ can be any value below 5, so it does not have a definite assignment. Using these two new definite assignments, all three formulas $a$ and $b$ occur in are replaced with their new versions.

By using configuration option `cpa.constraints.mergeType = SEP` or `JOIN`, either $\texttt{merge}^{sep}$, as used in [13], 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.

## 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 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 4.4.1) 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 general 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. Precision type. 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 implemention 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.

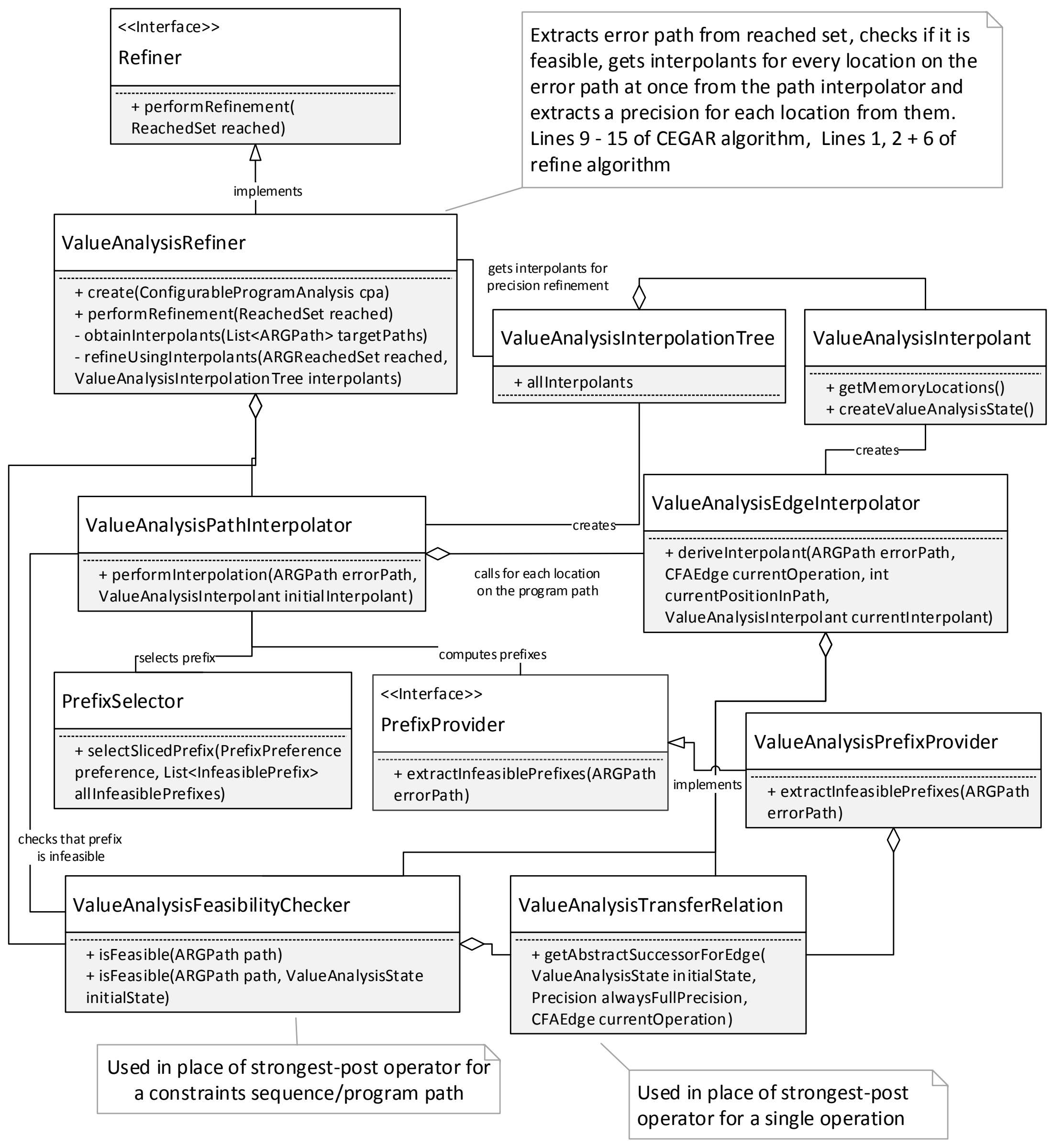
3. Interpolation algorithm with strongest-post operator and structure of the produced interpolant. Symbolic execution uses an interpolation algorithm with another behaviour and a different structure of the returned interpolant. This has to be considered in the `extractPrecision`, also.

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 7 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. 3) 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. 3).

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, CPAChecker-provides the option to resume the CPA algorithm 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 $l_{i-1}$ is the one before the first pair $(op_i, l_i)$ whose corresponding



<<Interface>>
Refiner
+ performRefinement(ReachedSet reached)
Extracts error path from reached set, checks if it is feasible, gets interpolants for every location on the error path at once from the path interpolator and extracts a precision for each location from them. Lines 9 - 15 of CEGAR algorithm, Lines 1, 2 + 6 of refine algorithm
implements
ValueAnalysisRefiner
+ create(ConfigurableProgramAnalysis cpa)
+ performRefinement(ReachedSet reached)
- obtainInterpolants(List<ARGPath> targetPaths)
- refineUsingInterpolants(ARGReachedSet reached, ValueAnalysisInterpolationTree interpolants)
gets interpolants for precision refinement
ValueAnalysisInterpolationTree
+ allInterpolants
ValueAnalysisInterpolant
+ getMemoryLocations()
+ createValueAnalysisState()
creates
ValueAnalysisPathInterpolator
+ performInterpolation(ARGPath errorPath, ValueAnalysisInterpolant initialInterpolant)
creates
calls for each location on the program path
ValueAnalysisEdgeInterpolator
+ deriveInterpolant(ARGPath errorPath, CFAEdge currentOperation, int currentPositionInPath, ValueAnalysisInterpolant currentInterpolant)
selects prefix
computes prefixes
PrefixSelector
+ selectSlicedPrefix(PrefixPreference preference, List<InfeasiblePrefix> allInfeasiblePrefixes)
<<Interface>>
PrefixProvider
+ extractInfeasiblePrefixes(ARGPath errorPath)
implements
ValueAnalysisPrefixProvider
+ extractInfeasiblePrefixes(ARGPath errorPath)
checks that prefix is infeasible
ValueAnalysisFeasibilityChecker
+ isFeasible(ARGPath path)
+ isFeasible(ARGPath path, ValueAnalysisState initialState)
ValueAnalysisTransferRelation
+ getAbstractSuccessorForEdge(ValueAnalysisState initialState, Precision alwaysFullPrecision, CFAEdge currentOperation)
Used in place of strongest-post operator for a constraints sequence/program path
Used in place of strongest-post operator for a single operation

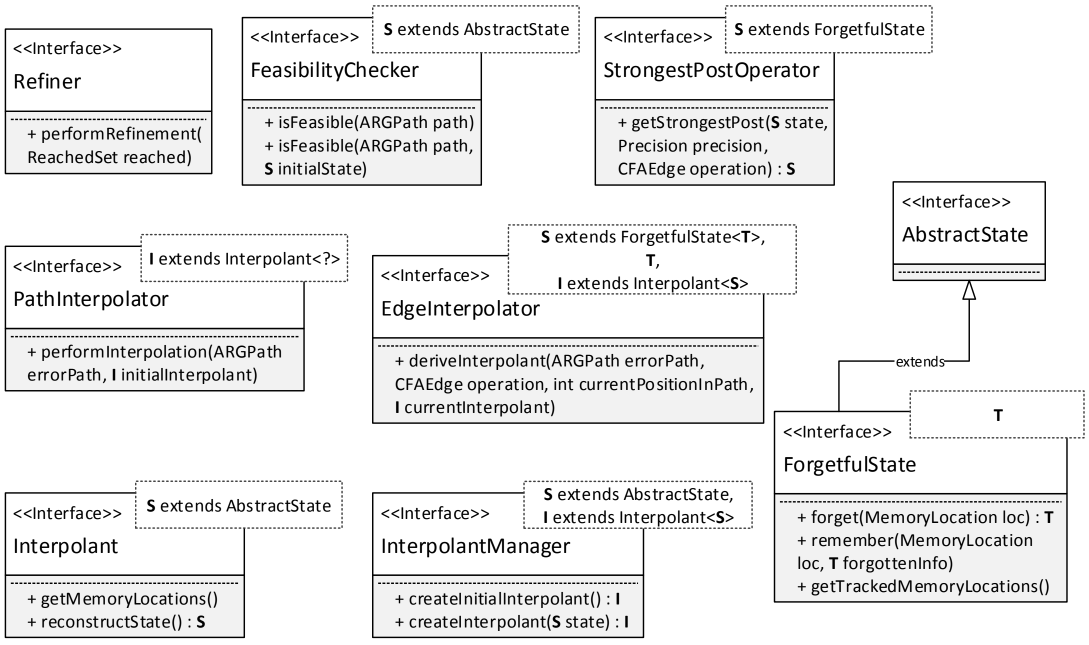
Figure 7: Structure of value analysis CPA refinement before refactoring

interpolant is not the empty constraint sequence (i.e. the location before the first location with a new precision). This restart strategy is called *pivot*. The option `cpa.value.refinement.restartStrategy = PIVOT` activates this behaviour instead of a full reset.[**?**]

A deviation from the `refine` algorithm (Alg. 5) is that the interpolants for all program locations on the error path are created by the `ValueAnalysisPathInterpolator` in one go, stored as a `ValueAnalysisInterpolationTree`. It basically represents Lines 3 – 5 of the refine algorithm, while `ValueAnalysisEdgeInterpolator` is used for concrete interpolation. The interpolation algorithm (Alg. 4) 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 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 `ValueAnalysisInterpolant`, so the strongest-post operator only has to be applyed 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 `ValueAnalysisFeasibilityChecker` is used for the feasibility check `isFeasible` (Alg. 3, Line 10). 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 `ValueAnalysisPrefixProvider` 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 [10] and will be used by us without modification. The algorithm for determining infeasible prefixes also uses the strongest-post operator. In the implementation, `ValueAnalysisPrefixProvider` 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



<<Interface>>
Refiner
+ performRefinement( ReachedSet reached)
<<Interface>>
FeasibilityChecker
S extends AbstractState
+ isFeasible(ARGPath path)
+ isFeasible(ARGPath path, S initialState)
<<Interface>>
StrongestPostOperator
S extends ForgetfulState
+ getStrongestPost(S state, Precision precision, CFAEdge operation) : S
<<Interface>>
AbstractState
<<Interface>>
PathInterpolator
I extends Interpolant<?>
+ performInterpolation(ARGPath errorPath, I initialInterpolant)
<<Interface>>
EdgeInterpolator
S extends ForgetfulState<T>, T, I extends Interpolant<S>
+ deriveInterpolant(ARGPath errorPath, CFAEdge operation, int currentPositionInPath, I currentInterpolant)
extends
<<Interface>>
ForgetfulState
T
+ forget(MemoryLocation loc) : T
+ remember(MemoryLocation loc, T forgottenInfo)
+ getTrackedMemoryLocations()
<<Interface>>
Interpolant
S extends AbstractState
+ getMemoryLocations()
+ reconstructState() : S
<<Interface>>
InterpolantManager
S extends AbstractState, I extends Interpolant<S>
+ createInitialInterpolant() : I
+ createInterpolant(S state) : I

Figure 8: Interfaces used in refinement

when structs or arrays occur.

### 6.1.2 Introduction of interfaces

Except for `ValueAnalysisRefiner` and `ValueAnalysisPrefixProvider`, 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 implementation of the value analysis refinement. These interfaces are displayed in Figure 8 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 readding 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 this type and the type 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 9. 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 + ConstraintsCPA

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 10 shows the structure of this refinement.

`SymbolicInterpolant` implements `Interpolant`. It stores information about abstract variable assignments and constraints, so it can be used for

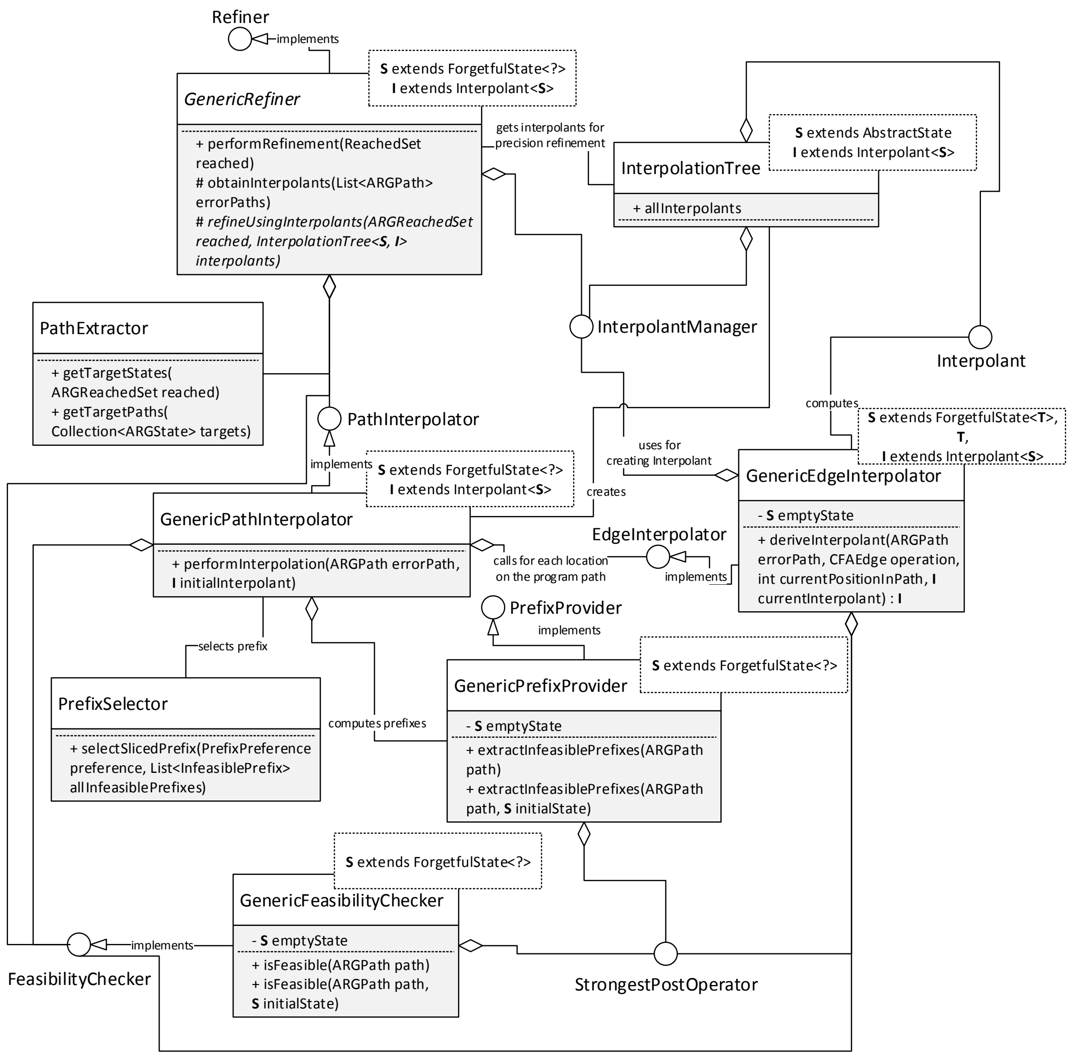
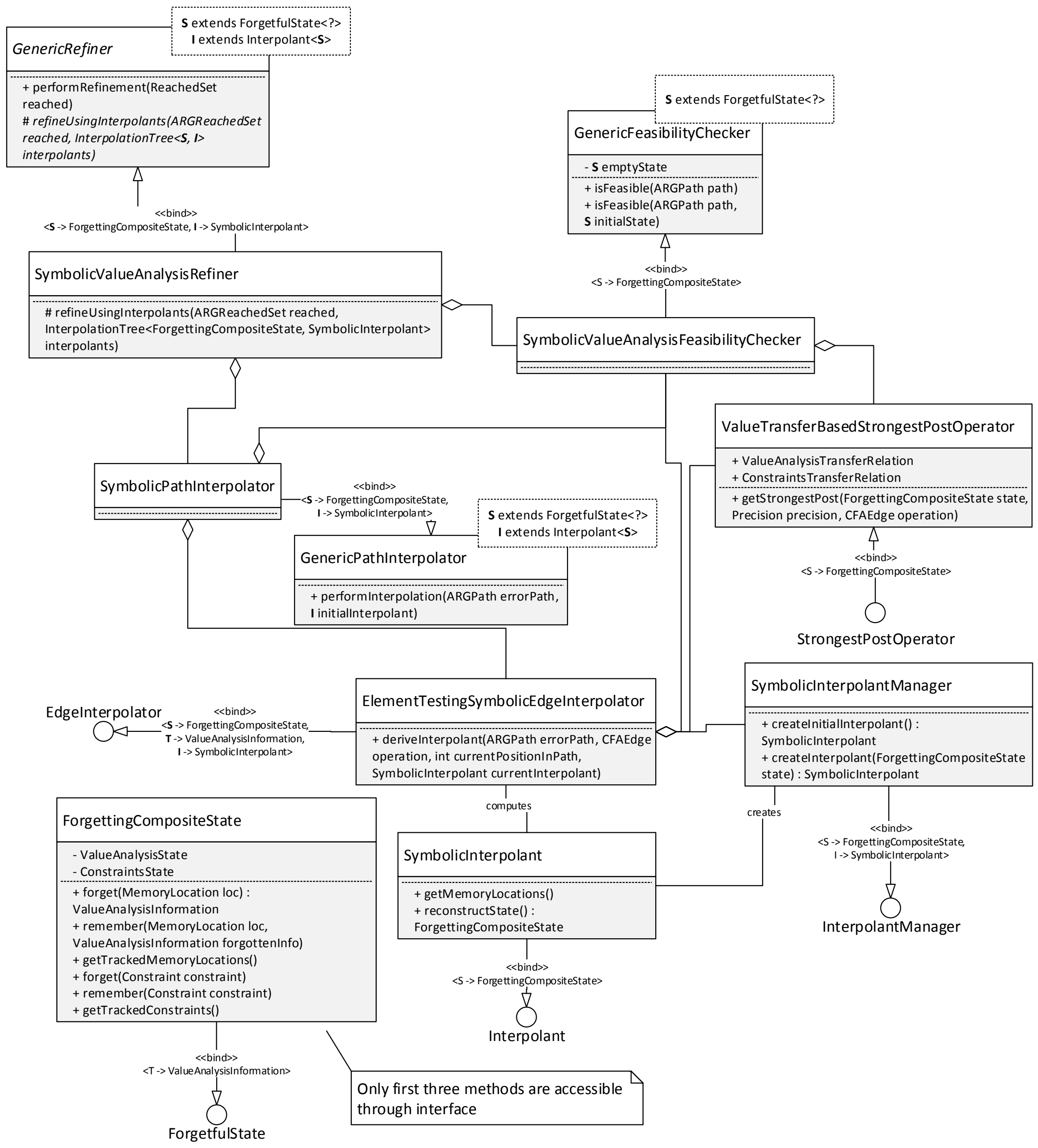

Refiner
implements
S extends ForgetfulState<?>
I extends Interpolant<S>
GenericRefiner
+ performRefinement(ReachedSet reached)
# obtainInterpolants(List<ARGPath> errorPaths)
# refineUsingInterpolants(ARGReachedSet reached, InterpolationTree<S, I> interpolants)
gets interpolants for precision refinement
InterpolationTree
S extends AbstractState
I extends Interpolant<S>
+ allInterpolants
PathExtractor
+ getTargetStates( ARGReachedSet reached)
+ getTargetPaths( Collection<ARGState> targets)
InterpolantManager
Interpolant
PathInterpolator
computes
S extends ForgetfulState<T>, T, I extends Interpolant<S>
uses for creating Interpolant
GenericEdgeInterpolator
- S emptyState
+ deriveInterpolant(ARGPath errorPath, CFAEdge operation, int currentPositionInPath, I currentInterpolant) : I
implements
S extends ForgetfulState<?>
I extends Interpolant<S>
creates
GenericPathInterpolator
+ performInterpolation(ARGPath errorPath, I initialInterpolant)
calls for each location on the program path
EdgeInterpolator
implements
PrefixProvider
implements
selects prefix
S extends ForgetfulState<?>
GenericPrefixProvider
- S emptyState
+ extractInfeasiblePrefixes(ARGPath path)
+ extractInfeasiblePrefixes(ARGPath path, S initialState)
PrefixSelector
+ selectSlicedPrefix(PrefixPreference preference, List<InfeasiblePrefix> allInfeasiblePrefixes)
computes prefixes
S extends ForgetfulState<?>
GenericFeasibilityChecker
- S emptyState
+ isFeasible(ARGPath path)
+ isFeasible(ARGPath path, S initialState)
implements
FeasibilityChecker
StrongestPostOperator

Figure 9: Structure of generic refinement procedure for abstract variable assignments

interpolating over both these types. `ForgettingCompositeState` implements `ForgetfulState`. It is the composition of `ValueAnalysisState` and `ConstraintsState` and provides methods for forgetting/remembering both their elements separately. This is necessary for interpolation, described below. `SymbolicInterpolantManager` is an `InterpolantManager` able to create `SymbolicInterpolants`. `ValueTransferBasedStrongest-PostOperator` is the implementation of the composite strongest-post operator using the value analysis transfer relation and constraints transfer



GenericRefiner
S extends ForgetfulState<?>
I extends Interpolant<S>
+ performRefinement(ReachedSet reached)
# refineUsingInterpolants(ARGReachedSet reached, InterpolationTree<S, I> interpolants)
<<bind>>
<S -> ForgettingCompositeState, I -> SymbolicInterpolant>
SymbolicValueAnalysisRefiner
# refineUsingInterpolants(ARGReachedSet reached, InterpolationTree<ForgettingCompositeState, SymbolicInterpolant> interpolants)
GenericFeasibilityChecker
S extends ForgetfulState<?>
- S emptyState
+ isFeasible(ARGPath path)
+ isFeasible(ARGPath path, S initialState)
<<bind>>
<S -> ForgettingCompositeState>
SymbolicValueAnalysisFeasibilityChecker
ValueTransferBasedStrongestPostOperator
+ ValueAnalysisTransferRelation
+ ConstraintsTransferRelation
+ getStrongestPost(ForgettingCompositeState state, Precision precision, CFAEdge operation)
<<bind>>
<S -> ForgettingCompositeState>
StrongestPostOperator
SymbolicPathInterpolator
<<bind>>
<S -> ForgettingCompositeState,
I -> SymbolicInterpolant>
S extends ForgetfulState<?>
I extends Interpolant<S>
GenericPathInterpolator
+ performInterpolation(ARGPath errorPath, I initialInterpolant)
EdgeInterpolator
<<bind>>
<S -> ForgettingCompositeState,
T -> ValueAnalysisInformation,
I -> SymbolicInterpolant>
ElementTestingSymbolicEdgeInterpolator
+ deriveInterpolant(ARGPath errorPath, CFAEdge operation, int currentPositionInPath, SymbolicInterpolant currentInterpolant)
SymbolicInterpolantManager
+ createInitialInterpolant() : SymbolicInterpolant
+ createInterpolant(ForgettingCompositeState state) : SymbolicInterpolant
computes
creates
<<bind>>
<S -> ForgettingCompositeState,
I -> SymbolicInterpolant>
InterpolantManager
ForgettingCompositeState
- ValueAnalysisState
- ConstraintsState
+ forget(MemoryLocation loc) : ValueAnalysisInformation
+ remember(MemoryLocation loc, ValueAnalysisInformation forgottenInfo)
+ getTrackedMemoryLocations()
+ forget(Constraint constraint)
+ remember(Constraint constraint)
+ getTrackedConstraints()
SymbolicInterpolant
+ getMemoryLocations()
+ reconstructState() : ForgettingCompositeState
<<bind>>
<S -> ForgettingCompositeState>
Interpolant
<<bind>>
<T -> ValueAnalysisInformation>
ForgetfulState
Only first three methods are accessible through interface

Figure 10: Structure of refinement procedure for symbolic execution

relation, described in Section 4.4.1.

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

#### 6.2.1 Extract precision from predicate refinement

# 7 Evaluation

## 7.1 Specification of evaluation environment

## 7.2 Evaluation on symbolic value analysis without CEGAR

### 7.2.1 Different merge operators

### 7.2.2 Different less-or-equal operators

### 7.2.3 SAT checks at different locations

## 7.3 Evaluation of symbolic value analysis with CEGAR

### 7.3.1 Refine only value analysis

### 7.3.2 Refine first value analysis, then constraints

### 7.3.3 Refine first constraints, then value analysis

## 7.4 Changed impact of above mentioned options when using CEGAR

## 7.5 Comparison of Symbolic Value Analysis with CEGAR and without CEGAR

## 7.6 Problems CEGAR can't solve/newly creates

### 7.6.1 Many variables needed for proof can decrease performance

### 7.6.2 Problem of long/endless loops and path explosion because of these can't be solved

## 7.7 Comparison with basic Value Analysis and Predicate Analysis

# 8 Future work

# 9 Conclusion

# References

[1] Anand, S., Godefroid, P., and Tillmann, N. Demand-Driven Compositional Symbolic Execution. In *Proceedings of the 14th International Conference, TACAS* (2008), pp. 367–381.

[2] Ball, T., Podelski, A., and Rajamani, S. K. Boolean and Cartesian abstractions for model checking C programs. In *Proceedings of the 7th International Conference, TACAS* (2001), pp. 268–283.

[3] Beyer, D. Second Competition on Software Verification (Summary of SV-COMP 2013). In *Proc. TACAS* (2013), vol. 7795 of *LNCS*, Springer, pp. 594–609.

[4] Beyer, D. Status Report on Software Verification (Competition Summary SV-COMP 2014). In *Proc. TACAS* (2014), vol. 8413 of *LNCS*, Springer, pp. 373–388.

[5] Beyer, D. Software Verification and Verifiable Witnesses (Report on SV-COMP 2015). In *Proc. TACAS* (2015), LNCS, Springer. (To appear).

[6] Beyer, D., Gulwani, S., and Schmidt, D. A. Combining Model Checking and Data-Flow Analysis. 1–50.

[7] Beyer, D., Henzinger, T. a., and Théoduloz, G. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. *Computer Aided Verification 4590* (2007), 504–518.

[8] Beyer, D., and Keremoglu, M. E. CPAchecker: A tool for configurable software verification. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 6806 LNCS* (2011), 184–190.

[9] Beyer, D., and Löwe, S. Explicit-state software model checking based on CEGAR and interpolation. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE)* (2013), pp. 146–162.

[10] Beyer, D., Löwe, S., and Wendler, P. Sliced Path Prefixes.

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

[12] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM 50*, 5 (2003), 752–794.

[13] Lemberger, T. Symbolic Execution in CPAchecker. Tech. rep., Chair of Software Systems, University of Passau, Passau, 2015.

[14] Myers, G. J., Sandler, C., and Badgett, T. *The Art of Software Testing*, 3rd ed. John Wiley & Sons, 2011.