UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik

Bachelor's Thesis
in Internet Computing

Octagon-Based Software Verification with CPAchecker

Thomas Stieglmaier

Supervisor:
Prof. Dr. Dirk Beyer

July 10, 2014

Abstract

This work introduces a formalism for a configurable program analysis (CPA) based on the octagon abstract domain. Several different configurations of this CPA and how it can be used to analyze programs in a time- and memory-efficient manner are discussed. The evaluation of these approaches is done on two implementations of octagon-based CPAs in CPACHECKER, a CPA using the Octagon Abstract Domain Library and a CPA using the APRON library. Both CPAs are compared to other analyses implemented in CPACHECKER and other tools. Overall, octagon-based CPAs are not as memory-efficient as an explicit-value analysis, but they perform strictly better on programs which rely on inter-variable relations.

Contents

1 Introduction **8**

1.1 Motivation 8

1.2 Structure of this Bachelors Thesis 9

2 Background **10**

2.1 Program Representation 10

2.2 Configurable Program Analysis with Dynamic Precision Adjustment 11

2.2.1 Formalism of a CPA+ 12

2.2.2 The Reachability Algorithm 14

2.2.3 Composite Program Analysis 16

2.3 The CPACHECKER Framework 17

2.3.1 Basic Architecture 17

2.3.2 Composite CPAs in CPACHECKER 18

2.3.3 Explicit-Value Analysis 19

2.3.4 Counterexample Check and Sequential Combination of Analyses 19

2.4 Counterexample-Guided Abstraction Refinement for Analyses with Explicit Values 20

2.4.1 Example 21

2.4.2 Interpolation and Precision Refinement 23

2.5 The Octagon Abstract Domain 24

2.5.1 Formalism 25

2.5.2 The Octagon Abstract Domain Library 27

2.5.3 APRON Library Architecture & Interface 29

2.5.4 PAGAI 29

3 Octagon-Based Software Verification **30**

3.1 The Octagon CPA 30

3.2 Influence of the merge Operator on the Analysis 32

3.3 Advanced Algorithms for the Octagon CPA 35
3.3.1 The Octagon CPA and CEGAR 35
3.3.2 Sequential Combination of different Configurations of the Octagon CPA 36

4 Implementation of Octagon-Based CPAs 38
4.1 A CPA Using the Octagon Abstract Domain Library 38
4.1.1 Architecture Overview 39
4.1.2 Specific Configuration Options of the OADL CPA 41
4.2 A CPA Using the APRON Library 41
4.2.1 Architecture Overview 42
4.2.2 Specific Configuration Options for the APRON CPA 44
4.3 Common Parts of the OADL CPA and the APRON CPA 44
4.3.1 The CEGAR Implementation 45
4.3.2 Configuration Options 46
4.4 Comparison of the OADL CPA and the APRON CPA Regarding the Programming Effort 48

5 Evaluation 50
5.1 Benchmark Programs 50
5.2 Configurations 51
5.3 Evaluation Environment 53
5.4 Performance Evaluation of the OADL CPA and the APRON CPA . . 53
5.4.1 Evaluation of Integer-Related Programs 54
5.4.2 Evaluation of Float-Related Programs 60
5.4.3 Custom Programs for Showing the Abilities of the OADL CPA 61
5.4.4 Conclusion of the Performance Evaluation 62
5.5 Restrictions and Challenges 63

6 Conclusion 65
6.1 Summary 65
6.2 Future Work 65

Bibliography 67

List of Algorithms

1 $CPA{+}(\mathbb{D}, R_0, W_0)$ algorithm, adapted from Beyer and Löwe [BL13] . 15

2 $CEGAR(\mathbb{D}, e_0, \pi_0)$, taken from Beyer and Löwe [BL13] 22

List of Figures

2.1 The CFA for the program in Listing 2.1 11
2.2 CPACHECKER Architecture, after Beyer and Keremoglu [BK11] . . . 17
2.3 CPACHECKER CPA design, after Beyer and Keremoglu [BK11] 18
2.4 Octagon Abstract Domain – A comparison of some numerical domains regarding over-approximation, after Miné [Min06] 24

3.1 The Octagon CPA and CEGAR 36
3.2 Sequential combination of different configurations of the Octagon CPA 36

4.1 An example octagon 40

5.1 A plot of quantile functions of the configurations **oadlSep**, **oadl-refiner**, **eva-basic** and **eva-refiner** 56
5.2 A plot of quantile functions of the configurations **oadlWidening-cex**, **oadl-refiner** and **oadl-seq-W-R-100** 59

List of Tables

2.1 Constraints and their equivalent as octagonal constraint, taken from Miné [Min06] . . . 26
2.2 Difference-Bound Matrice for the octagon from Figure 2.4 . . . 28

3.1 Results of an analysis with the Octagon CPA when using different merge operators . . . 33

5.1 Overall Performance of the Analyses on the SV-COMP Benchmarks . 55
5.2 Average Number of refinements, states in the set reached and memory consumption for certain configurations, only successfully-analyzed verification tasks are considered . . . 57
5.3 Number of created states, states in the set reached and average memory consumption for certain configurations . . . 58
5.4 Overall performance of the analyses on the SV-COMP benchmarks . 60
5.5 Overall Performance of the Analyses on the SV-COMP Benchmarks . 61

1 Introduction

Bugs exist since the invention of programming languages. Finding them is often much more work than writing the program itself. One possibility to find bugs is writing tests for a program, but tests can only cover a small range of the possible states that can occur in a program. And as software gets more and more important in nowadays life, the need for correct and reliable software is growing. The solution is a formal approach which guarantees that a program is safe or unsafe with regard to a certain specification.

1.1 Motivation

Regarding the results of the International Competition on Software Verification from 2014 [Bey14] one can see several tools which aim at providing a solution to the problem mentioned in the last paragraph.

CPACHECKER is the second winner in the category "Overall"[1], and because of its software-design, it is the ideal platform for implementing new analyses. Along with the explicit-value and predicate analysis, the two main analyses in CPACHECKER, there are several others existing. Apart from the predicate analysis none of the widely used analyses is currently capable of handling relations between variables. This work introduces a new analysis in CPACHECKER which is able to do so. The octagon abstract domain [Min06] handles linear constraints between numeric variables. For that reason it is more precise than an explicit-value analysis. Through the limitation to maximally two variables per linear constraint the octagon abstract domain is also limited in terms of the computation time. Thus it is more efficient than a predicate analysis. Furthermore it is also relatively lightweight compared to

[1] The complete results can be seen at the following website: `http://sv-comp.sosy-lab.org/2014/results/index.php`

other relational domains like the polyhedron abstract domain [Min06]. Therefore it is a promising candidate for combining it with other analyses in CPACHECKER.

1.2 Structure of this Bachelors Thesis

At first all the necessary background information is given. This includes describing the formal CPA algorithm, the octagon abstract domain (and its implementations in the Octagon Abstract Domain Library and the APRON library), CPACHECKER itself and at last the Counterexample-Guided Abstraction Refinement algorithm.

The background section is followed by a formal definition of an octagon-based CPA and an in detail description of additional features. Implementations of this formal CPA definition are introduced in the next step and afterwards evaluated with regard to their efficiency and effectiveness:

- with different kinds of programs, as included in the benchmark set of the International Competition on Software Verification, but also with artificial programs of other benchmark suits and some custom benchmarks,
- compared to other state-of-the-art analyses done with CPACHECKER, and
- compared to other software-verification tools using octagons.

The evaluation also covers the strengths and weaknesses of different configurations of these CPAs, for example with alternative merge strategies or advanced analysis algorithms. Finally a summary of the whole work is provided, and there is an outlook on features which could improve octagon-based CPAs even more.

2 Background

In this chapter all theoretical concepts that are necessary to create a configurable program analysis based on the octagon abstract domain get introduced. Furthermore some advance concepts, such as counterexample-guided abstraction refinement or parts of conditional model checking will be described.

2.1 Program Representation

A program is represented by a *control-flow automaton* (CFA) [BHT07]. This is a graph that consists of a set L of nodes / locations, modeling the program counter (pc), a set $G \subseteq L \times Ops \times L$ of edges which represent the control-flow and an initial location pc_0 (the program entry point). We limit the operations at control-flow edges to assumptions and assignments, consisting of linear expressions. Variables may only represent unbounded integers and rationals. The *concrete state* c of a program assigns to each variable from the set $X \cup \{pc\}$ a value. Let C be the set of all concrete states, and let every edge $g \in G$ define the transition relation $\xrightarrow{g} \subseteq C \times \{g\} \times C$. Then by unifying all edges the complete transition relation $\rightarrow = \bigcup_{g \in G} \xrightarrow{g}$ is created. On a CFA we now define reachability in the following way:

Reachability. If there exists a chain of concrete states $\langle c_0, c_1, ..., c_n \rangle$ with the requirement that $\forall i : 1 \leq i \leq n \implies c_{i-1} \rightarrow c_i$ and a region r such that $c_0 \in r$, then we call the state c_n reachable from the region r.

Example. In Figure 2.1 one can see the CFA for the program from Listing 2.1. While each arrow represents a control-flow edge of G, the circles represent the program locations L. The red circle with index 1 is the initial program location pc_0. Blue circles mark end points of the CFA.

```
1     int main(void) {
2       int flag = nondet_int();
3       int ticks = 0;

5       while (!systemCall()) {
6         ticks = ticks + 1;
7       }

9       if (flag <= 0) { return 0; }
10      if (flag > 0) { return 0; }

12      __VERIFIER_error();
13      return 0;
14    }
```

Listing 2.1: Example program 1

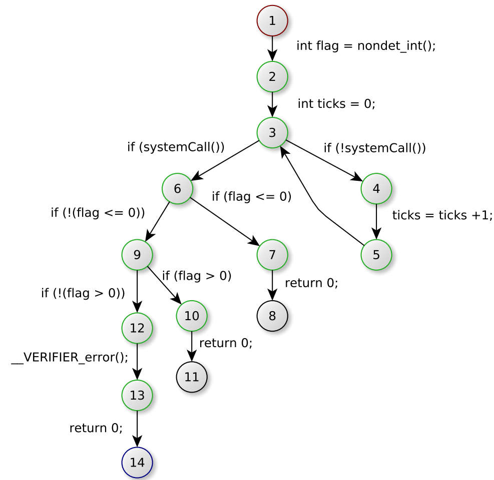

Figure 2.1: The CFA for the program in Listing 2.1

2.2 Configurable Program Analysis with Dynamic Precision Adjustment

In automated software verification two main approaches are model checking and program (data-flow) analysis [BHT07]. Both methods have their downsides. While software model checkers have an exploding state space for large programs, program or data-flow analyzers are usually path-insensitive. This means that in model checking, the amount of states created can often only be handled through abstraction and over-approximation. For that reason false alarms are a major problem of this approach. The path insensitivity of data-flow analyzers (they merge states with equal locations) makes the analysis imprecise regarding join points in programs.

By combining both approaches one can, on the one hand, reduce the state space drastically by merging at least some states, compared to a pure model checker. And on the other hand, by merging only states with certain attributes the accuracy will not suffer as much as with a pure program analysis.

The original configurable program analysis [BHT07] consists of four components, which influence the cost and accuracy of the analysis. By adding a so called precision to each abstract state and providing a precision adjustment function the CPA with dynamic precision adjustment (CPA+) was created [BHT08]. As an example for the

usage of precision one can imagine having a coarse-grained precision which specifies to only track certain variables in the analysis process. An other option would be to specify how detailed a variable should be tracked, depending on the progress of an analysis. The ability to adjust the precision at run time has a great influence on the verification performance. Two extreme examples are either switching off the tracking of any variable at all, or tracking every variable with the highest detail possible.

2.2.1 Formalism of a CPA+

A CPA+ [BHT08] $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ consists of six parts, an abstract domain D, a set Π of precisions, a transfer relation $\rightsquigarrow$, a merge operator merge, a termination check stop and a precision adjustment function prec. Those parts will be briefly described in the following paragraph:

- The *abstract domain* $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of three components. The first two parts are a set C of concrete states and a semi-lattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ consisting of
  - a potentially infinite set E of elements (those will be called abstract states later on),
  - a top element $\top$ and a bottom element $\bot$ with $\top, \bot \in E$,
  - a preorder $\sqsubseteq \subseteq E \times E$,
  - and a total function $\sqcup : E \times E \rightarrow E$ (which is the join operator).

  The third part is a concretization function $[\![\cdot]\!] : E \rightarrow 2^C$. It assigns its meaning to each abstract state, thus the set of concrete states it is representing. For soundness the abstract domain has to follow two requirements:

  1. $[\![\top]\!] = C$ and $[\![\bot]\!] = \emptyset$
  2. $\forall e, e' \in E : [\![e \sqcup e']\!] \supseteq [\![e]\!] \cup [\![e']\!]$ (either the join operator is precise or it over-approximates)

- The set Π models the possible precisions of the abstract domain D. Let e be an abstract state and π a precision. We call a pair (e, π) *the abstract state e with precision* π. Moreover all operators on the abstract domain are parametric in the precision.

- The *transfer relation* $\rightsquigarrow \subseteq E \times G \times E \times \Pi$ assigns for a CFA edge $g \in G$ all possible new abstract states e' with precision π to every abstract state $e \in E$. If $(e, g, e', \pi) \in \rightsquigarrow$ then we write $e \overset{g}{\rightsquigarrow} (e', \pi)$ and if an edge g exists with $e \overset{g}{\rightsquigarrow} (e', \pi)$ we write $e \rightsquigarrow (e', \pi)$. For soundness the transfer relation has to follow the requirement that $\forall e \in E, g \in G, \pi \in \Pi : \bigcup_{e \rightsquigarrow (e', \pi)} [\![e']\!] \supseteq \bigcup_{c \in [\![e]\!]} \left\{ c' | c \xrightarrow{g} c' \right\}$ (the transfer relation over-approximates all operations for every fixed precision).

- The *merge operator* $\mathsf{merge} : E \times E \times \Pi \to E$ merges the information of two abstract states. Soundness is achieved by the requirement that

$$\forall e, e' \in E, \pi \in \Pi : e' \sqsubseteq \mathsf{merge}(e, e', \pi)$$

  (the result may only be equal to or more abstract than the second parameter). This means that depending on e and the precision π the merge result can be anything between e' and $\top$ and furthermore that the merge operator is not commutative. Although the merge operator is not the same as the join operator $\sqcup$ from the semi-lattice, it can be based on it. The two most commonly used merge operators are $\mathsf{merge}^{sep}(e, e') = e'$ and $\mathsf{merge}^{join}(e, e') = e \sqcup e'$.

- The *termination check* $\mathsf{stop} : E \times 2^E \times \Pi \to \mathbb{B}$ checks whether the set of abstract states R, given as second parameter, covers the state with its precision given as first and third parameter. To ensure soundness the termination check has to satisfy the requirement

$$\forall e \in E, R \subseteq E, \pi \in \Pi : \mathsf{stop}(e, R, \pi) = true \Rightarrow [\![e]\!] \subseteq \bigcup_{e' \in R} [\![e']\!] \,.$$

  This means that if an abstract state e is covered by the set R also every concrete state represented by e corresponds to an abstract state from the set R. Equivalent to the merge operator, the termination check is not the same as the preorder $\sqsubseteq$ of the semi-lattice, but can be based on it. For example the termination check can iterate over all elements from the second parameter and search for a single element that contains ($\sqsubseteq$) the first parameter. Later on, the termination check $\mathsf{stop}^{sep}(e, R) = (\exists e' \in R : e \sqsubseteq e')$ will be used.

- The *precision adjustment function* $\mathsf{prec} : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$ creates a new abstract state with precision for a given abstract state with precision and a given set of abstract states with precisions. During the precision change the

prec function may also perform a widening of the abstract state, thus it is able to decrease or increase the precision of abstract states. The requirement which has to be fulfilled to ascertain soundness is, that

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

2.2.2 The Reachability Algorithm

In the last section all necessary members of a CPA+ were introduced. The algorithm which computes a set of abstract states (for example an over-approximation of the set of reachable concrete states) for a given CPA+ and an initial abstract state is a reachability algorithm [BHT08]. During the execution of this algorithm two sets get updated permanently:

- the set reached where all found reachable states are stored, and
- the set waitlist where all abstract states which have yet been found but not processed (frontier) are stored.

The reachability algorithm of the CPA+ computes a set of reachable abstract states with accompanying precision out of an initial abstract state with precision. The first step during the abstract successor computation is the precision adjustment with the prec function, thus the precision may change dynamically from abstract state to abstract state. Afterwards each successor state with precision is combined with every abstract state with precision from the set reached using the given merge operator. If additional information has been added, such that the old abstract state with precision is less abstract than the new abstract state with precision, it is replaced by the new one in the set reached. If the state with precision resulting from the merge step is not covered by any state in the set reached it is added to both, the set reached and the set waitlist.

In order to have an algorithm that may already be used for counterexample-guided abstraction refinement (c.f. Section 2.4) we adapt the input parameters of the CPA+ algorithms, such that instead of an initial abstract state with precision, a set R_0 of abstract states with precision is used. Additionally a subset $W_0 \subseteq R_0$ of frontier abstract states with precision has to be given as parameter [BL13]. Algorithm 1 is the resulting reachability algorithm.

Algorithm 1 $CPA{+}(\mathbb{D}, R_0, W_0)$ algorithm, adapted from Beyer and Löwe [BL13]

Input : a configurable program analysis $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$,
a set $R_0 \subseteq (E \times \Pi)$ of abstract states with precision, and
a subset $W_0 \subseteq R_0$ of frontier abstract states with precision,
where E denotes the set of elements of the semi-lattice of D

Output : a set of reachable abstract states with precision

Variables : a set reached of elements of $E \times \Pi$,
a set waitlist of elements of $E \times \Pi$

```
waitlist := W₀
reached := R₀
while waitlist ≠ ∅ do
    choose (e, π) from waitlist;
    waitlist := waitlist \ {(e, π)};
    for each e' with e ⇝ (e', π) do
        (ê, π̂) := prec(e', π, reached); // precision adjustment
        if isTargetState(ê) then
            return (reached ∪ {(ê, π̂)}, waitlist ∪ {(ê, π̂)});
        end
        for each (e'', π'') ∈ reached do
            e_new := merge(ê, e'', π̂); // combine with existing abstract state
            if e_new ≠ e'' then
                waitlist := (waitlist ∪ {(e_new, π̂)}) \ {(e'', π'')};
                reached := (reached ∪ {(e_new, π̂)}) \ {(e'', π'')};
            end
        end
        if ¬ stop(ê, {e | (e, ·) ∈ reached}, π̂) then
            // add new abstract state
            waitlist := waitlist ∪ {(ê, π̂)};
            reached := reached ∪ {(ê, π̂)};
        end
    end
end
return (reached, ∅);
```

2.2.3 Composite Program Analysis

The combination of several configurable program analyses [BHT07] is a composite program analysis[1] $\mathcal{C} = (\mathbb{D}_1, ..., \mathbb{D}_n, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times)$ with $n \in \mathbb{N}$. It consists of a finite amount of CPAs and the combined parts:

- a set of precisions $\Pi_\times$,
- a transfer relation $\rightsquigarrow_\times$,
- the merge operator $\mathsf{merge}_\times$,
- the stop operator $\mathsf{stop}_\times$,
- and the prec operator $\mathsf{prec}_\times$.

The four composites are any expressions over the components of the involved CPAs $(\Pi_i, \rightsquigarrow_i, \mathsf{merge}_i, \mathsf{stop}_i, \mathsf{prec}_i, [\![\cdot]\!]_i, E_i, \top_i, \bot_i, \sqsubseteq_i, \sqcup_i)$ with $i \in [1; n]$, as well as the strengthening operator $\downarrow: \times_{i=1}^{n} E_i \to E_1$ and the comparison operator $\preceq \subseteq \times_{i=1}^{n} E_i$. Strengthening is used for computing a stronger element from the lattice set E_1 by using the information from an element of the lattice sets $E_2 ... E_n$, thus $\downarrow (e_1, ..., e_n) \sqsubseteq e_1$ has to be fulfilled. The comparison operator allows us to compare elements of different lattices.

For a given composite analysis $\mathcal{C} = (\mathbb{D}_1, \mathbb{D}_2, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$ the CPA $\mathbb{D}_\times = (D_\times, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$ can be constructed. The product precision is defined by $\Pi_\times = \Pi_1 \times \Pi_2$. Equally, the components of the product domain $D_\times = D_1 \times D_2 = (C, \mathcal{E}_\times, [\![\cdot]\!]_\times)$ are then defined by the

- product lattice $\mathcal{E}_\times = \mathcal{E}_1 \times \mathcal{E}_2 = (E_1 \times E_2, (\top_1, \top_2), (\bot_1, \bot_2), \sqsubseteq_\times, \sqcup_\times)$ with $(e_1, e_2) \sqsubseteq_\times (e_1', e_2')$ iff $e_1 \sqsubseteq_1 e_1'$ and $e_2 \sqsubseteq_2 e_2'$, and $(e_1, e_2) \sqcup_\times (e_1', e_2') = (e_1 \sqcup_1 e_1', e_2 \sqcup_2 e_2')$
- and the product concretization function $[\![\cdot]\!]_\times$ in such a way, that $[\![(d_1, d_2)]\!]_\times = [\![d_1]\!]_1 \cap [\![d_2]\!]_2$ is met.

[1] This is still a CPA.

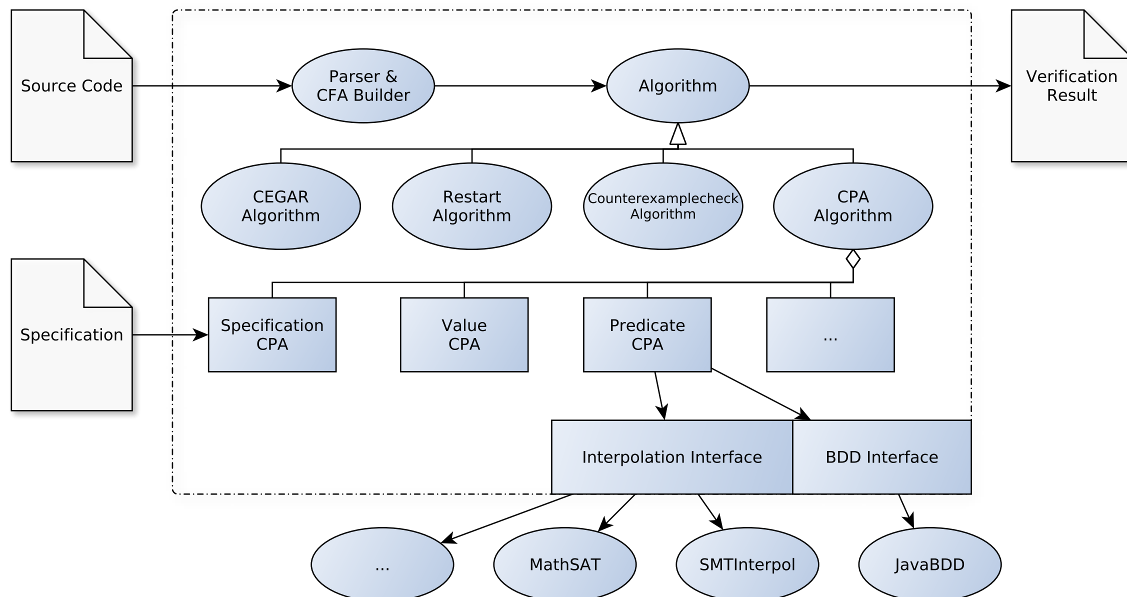

Figure 2.2: CPACHECKER Architecture, after Beyer and Keremoglu [BK11]

2.3 The CPACHECKER Framework

CPACHECKER[2] is a free and open-source software verification framework published under the Apache 2.0 license. It is the successor of the model-checker BLAST [BHJM05]. CPACHECKER is based on the concepts of configurable program analysis and dynamic precision adjustment [BK11]. The program analysis is performed by the implemented CPAs[3]. These CPAs can be combined freely, either for parallel usage at the same time (c.f. Section 2.2.3) or for a sequential usage (c.f. Section 2.3.4). The focus of CPACHECKER lies on the evaluation of programs written in C.

2.3.1 Basic Architecture

The basic architecture of CPACHECKER [BK11] is shown in Figure 2.2. CPACHECKER uses the parser of the Eclipse CDT project[4]. Subsequently the CFA (c.f. Section 2.1)

[2] More information and the sources can be found on the website of CPACHECKER: `http://cpachecker.sosy-lab.org/`

[3] Although it would be more correct to write CPA+ in the scope of CPACHECKER all implemented analyses are called CPAs, disregarding whether they are using dynamic precision adjustment or not.

[4] More information about the Eclipse CDT can be found here: `http://www.eclipse.org/cdt/`

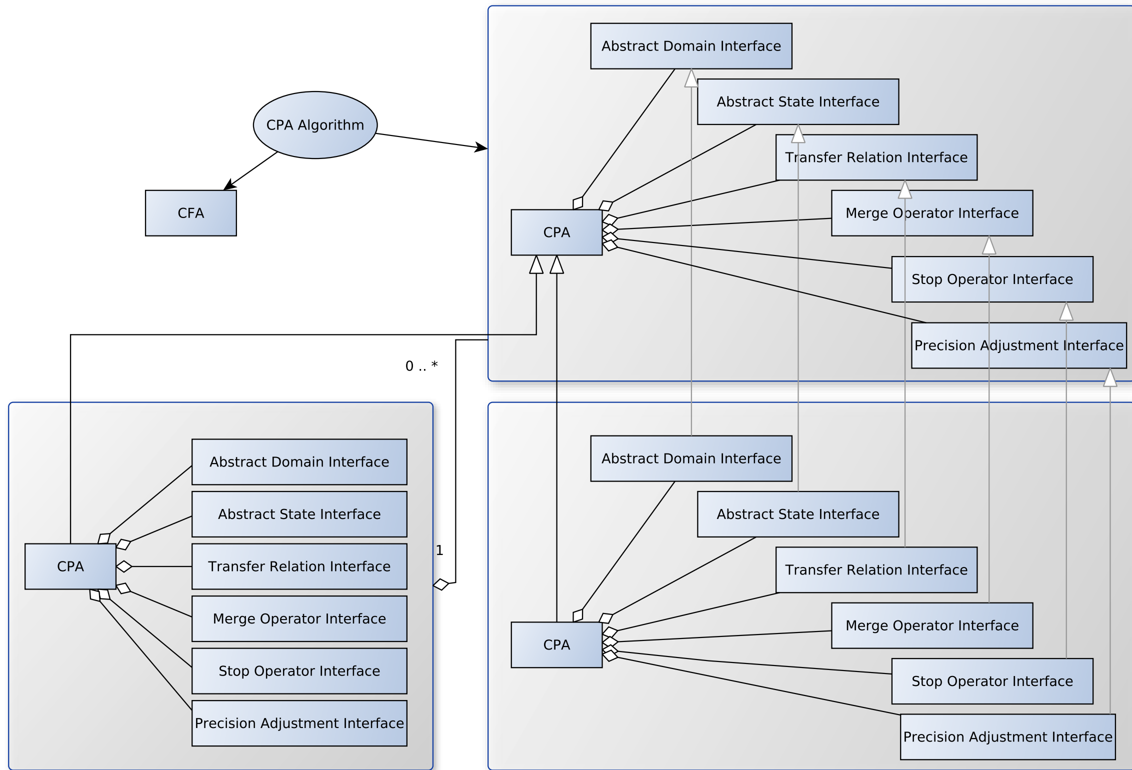

Figure 2.3: CPAchecker CPA design, after Beyer and Keremoglu [BK11]

is created, and then the CPA+ algorithm computes the result. This algorithm is the core of CPAchecker, it performs the reachability analysis (see Algorithm 1), and works on an instance of the abstract type CPA (so it doesn't know anything about the actual CPA). For that reason, every distinct CPA must implement the CPA interface (c.f. Figure 2.3) and all belonging methods.

2.3.2 Composite CPAs in CPAchecker

Through the concept of having a composite analysis consisting of several CPAs, which was introduced in Section 2.2.3, we are able to split up some analyses implemented in CPAchecker into several CPAs and then combine them on demand. For example, most analyses need a call stack and have to track the program counter, in order to be able to use the merge or stop operators properly. Thus, instead of implementing call stack and location-aware CPAs over and over, one can now create a CPA which models the call-stack and one CPA which tracks the program counter. These can then be reused together with every other CPA implementation which is in need of a call-stack and location-awareness.

2.3.3 Explicit-Value Analysis

As mentioned before, there are several different analyses and configurations which can be used with CPACHECKER. The explicit-value analysis works on distinct values for program variables[BL13]. As this analysis needs to be location-aware, it is constructed from a composite CPA which includes a CPA tracking the program locations, and a CPA for the explicit values. The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ for the Explicit Value CPA consists of a set C of concrete states, the semi-lattice $\mathcal{E} = (V, \top, \bot, \sqsubseteq, \sqcup)$ and the concretization function $[\![\cdot]\!] : V \rightarrow 2^C$. V is the set of abstract variable assignments $V = (X \rightarrow \mathcal{R})$, where X is the set of all variables and $\mathcal{R}$ is denoted by $\mathcal{R} = \mathbb{R} \cup \{\top, \bot\}$. In contrast to the $\bot$ element, which denotes that there is no value assignment possible, the $\top$ element has no specific value for any variable. By $v \sqsubseteq v'$ the partial order $\sqsubseteq \subseteq V \times V$ is defined when $\forall x \in X : v(x) = v'(x) \vee v(x) = \bot \vee v'(x) = \top$. Thus, the explicit-value analysis is not able to infer any information about the relations between program variables. The operators and functions that were not mentioned in the last paragraph, and more detailed information about the explicit-value analysis can be found in related literature [BL13].

2.3.4 Counterexample Check and Sequential Combination of Analyses

For a better control of the verification performance the dynamic precision adjustment was introduced in Section 2.2. Also by using CEGAR (c.f. Section 2.4) or different merge operator strategies the efficiency and effectiveness of the analyses can be influenced. What is missing, is the ability to reuse the results and additional information from previous verification runs. The counterexample check and starting another analysis after receiving a certain result, are part of the conditional model checking [BW13], which enables us to reuse exactly this information and results.

In contrast to traditional model checking, where the result of a verification run is either "safe" or "unsafe" (if a result was produced), conditional model checking puts out a *condition* Ψ that shows which conditions have to be fulfilled that the analyzed program satisfies a given specification. By producing such conditions in case of failures the consumed resources are not wasted. For example in case of a timeout the model checker could summarize the part of the program which was ver-

ified successfully in the outputted condition, declaring that as long as the program execution stays within this part, the program is safe. For a complete analysis “safe” is represented by $\Psi = true$ and “unsafe” is represented by $\Psi = false$.
Whereas the first verification run of a model checker starts per default with $false$ as input condition, in a sequential combination the second verification run of a model checker may get the condition outputted by the first model checker as input condition.
The **restart algorithm** used in this bachelor's thesis is such a sequential combination of verification tasks. Whenever the result of a verification run is not *true* or $false$, the next configuration is started with the input condition $false$ and has to verify the program again.
By using a previously found error path as input condition, a **counterexample check** can be realized. This counterexample check can either be done with the same analysis or with another analysis, also one from another model checking tool. By using a counterexample check one can also enhance the restart algorithm that was introduced before. For example by using a quite imprecise analysis which produces fast results but lots of false negatives at first. The feasibility of the error paths of these false negatives is then rechecked, and if the error path is infeasible, instead of a $false$ result, a condition is reported which states that the analysis was not complete. As a consequence, the restart algorithm does not only consider the next specified model checker when no result was outputted due to for example timeouts, but also due to impreciseness of analyses. Because of limiting the control flow to a certain path, checking a counterexample is easier and produces less states than a full analysis of a program.

2.4 Counterexample-Guided Abstraction Refinement for Analyses with Explicit Values

If a variable $x \in X$ is not necessary for proving a program safe with regard to a certain specification, it could be omitted in the abstract successor computation. Finding such unnecessary variables is done iteratively by a technique called counterexample-guided abstraction refinement (CEGAR) [BL13]. It is based on three concepts:

1. a precision, determining the tracked variables, and thus the current abstraction level,

2. a feasibility check for checking the validity of a found error path,
3. and a refinement function that creates a new precision out of an infeasible error path, such that the same error path cannot be reached again during the path exploration.

By starting with an initially empty precision, meaning not tracking variables at all, we will, in most cases, encounter infeasible counterexamples (paths through the control-flow graph that end in a state violating the given specification). The CEGAR algorithm for analyses with explicit values now computes the variables which have to be tracked additionally in order to avoid encountering the same infeasible counterexample once again. Then the analysis is restarted from a certain point with the refined precision. If the analysis is still too imprecise and due to a too coarse precision another infeasible counterexample is found, this loop of refining the precision and restarting the analysis is repeated over and over. The iterations of this loop last until either a feasible counterexample is found, or the program analyses terminates without finding a specification violation. The formal CEGAR algorithm, is shown in Algorithm 2.

To sum up the essence of the last paragraph, CEGAR is a technique where by continuous precision refinement the state space is kept as small as possible and as large as necessary. The computed states are mostly still more abstract, and at least as abstract as without CEGAR. For this reason CEGAR is a highly effective way to analyze programs with less effort than before. The following section shows this with an example.

2.4.1 Example

The example in Listing 2.1 illustrates the advantages of CEGAR compared to a traditional analysis, which stores explicit values for all variables. Due to the variable which is incremented in the while loop a traditional explicit-value analysis will discover a new abstract state with each loop iteration. The return value of the *systemCall* function is non-deterministic, and therefore the program analysis will never finish unrolling the loop, always discovering new abstract states.
When using CEGAR, at first the precision is empty, thus the error location will be reached. Then, through the refinement process we discover that the variable *flag* has to be tracked in order to prove that either the assumption $flag \leq 0$ or the assumption

Algorithm 2 $CEGAR(\mathbb{D}, e_0, \pi_0)$, taken from Beyer and Löwe [BL13]

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

Output : verification result safe or unsafe

Variables : set reached $\subseteq E \times \Pi$,
set waitlist $\subseteq E \times \Pi$,
error path $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$

```
reached := {(e0, π0)};
waitlist := {(e0, π0)};
while true do
    (reached, waitlist) := CPA(D, reached, waitlist);
    if waitlist = ∅ then return safe ;
    else
        σ := extractErrorPath(reached);

        // error path is feasible: report bug
        if isFeasible(σ) then return unsafe;

        // error path is not feasible: refine and restart
        else
            π := π ⋃ Refine(σ);
            reached := (e0, π);
            waitlist := (e0, π);
        end
    end
end
```

$flag > 0$ holds. Afterwards the analysis is restarted, and the error cannot be reached any more. Because *ticks* is still not tracked, unrolling the while loop creates no states which cannot be covered by those that were already found. So the analysis terminates in time, proving the program correct.

2.4.2 Interpolation and Precision Refinement

Craig interpolation [Cra57] is a technique from logics. It computes a new formula ψ for two contradicting formulas φ^- and φ^+ where $\varphi^- \wedge \varphi^+$ is unsatisfiable, called interpolant, that contains less information than the first formula, but is still contradicting the second formula. Thus it has to meet three requirements:

1. $\varphi^- \Rightarrow \psi$,
2. the conjunction $\psi \wedge \varphi^+$ has to be unsatisfiable, and
3. ψ may only contain symbols which occur in φ^- and φ^+.

In CEGAR such interpolants are mostly used for finding an appropriate precision refinement. The predicate analysis uses for example SMT solvers for the interpolation step. But also without SMT solvers, when having a precision which controls only which variables should be tracked, an interpolation can be done. This is the case for the explicit-value analysis [BL13], and will also be like that for the octagon-based analyses introduced in this work. While the predicate analysis has predicates / formulas for which the interpolation is made, the interpolation for analyses with explicit values works on counterexample paths. Such paths consist of abstract variable assignments (constraint sequences). For each path, a precision has to be determined, such that in future explorations the infeasible error path is eliminated.
This is done via iterating over all variables defined in the constraint sequence of the feasible part of the error path γ^-. γ^+ is the contradicting constraint sequence. Initially the strongest post-condition / the abstract successor of γ^- is computed and assigned to the abstract variable assignment v. Then in the interpolation process single variables are removed from v. If v still contradicts γ^+ the removed variable is not necessary for refuting the counterexample, otherwise the variable should occur in the interpolant.
From the interpolant a new precision is created in the *Refine* method, which is then unified with the old precision. The analysis should now be able to eliminate the error path with this **refined precision** in its next CEGAR algorithm iteration. The

complete idea of a refinement based on explicit interpolation is described in related literature [BL13].

2.5 The Octagon Abstract Domain

The octagon abstract domain is a weakly relational numerical domain [Min01]. It is based on Difference-Bound Matrices (c.f. Section 2.5.1) and therefore able to handle constraints of the form $\pm X \pm Y \leq c$ with X and Y being variables and c being a constant number in the range of the real numbers. Thus it can be seen as a more specific version of the polyhedron domain, which allows the representation of linear constraints with an arbitrary number of variables [CH78].

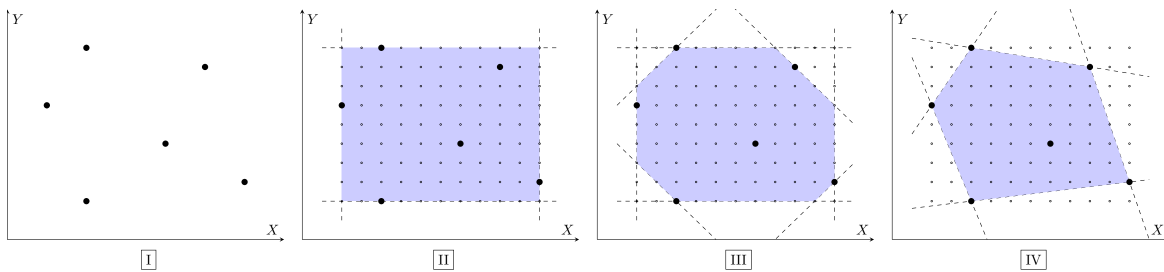

Figure 2.4: Octagon Abstract Domain – A comparison of some numerical domains regarding over-approximation, after Miné [Min06]

Figure 2.4 shows the differences between an interval domain, the octagon domain and a polyhedron domain while trying to get the most precise approximation of the values for X and Y. In the figure, (I) shows a set of points with distinct values for the variables X and Y. In all other graphs, the small dots in the colored areas mark points which are only considered as valid values for both variables due to over-approximation. While with intervals (II) only the lowest and the highest bound are taken into consideration, and therefore the resulting value range for X and Y is quite imprecise, the polyhedron domain (IV) is able to reduce the amount of over-approximated points[5]. Yet less precise than the polyhedron domain due to the bounded number of variables per constraint, the octagon domain (III) is also far less time and memory consuming, and still capable of representing more constraints

[5] In the polyhedron abstract domain the state space for X and Y could be reduced further through representing also relations with other variables. For the sake of this example two variables were considered sufficient.

than an interval domain. Its name refers to the maximum number of edges of the resulting shape in the coordinate system.

2.5.1 Formalism

The most important aspects about the octagon abstract domain are the internal representation through Difference-Bound Matrices and the way they are used for representing all kinds of constraints. The following two paragraphs give a brief overview over these parts.

Difference-Bound Matrices

Difference-Bound Matrices (DBMs) are a widely used representation in model-checking for constraints of the form $x - y \leq c$ and $\pm x \leq c$. More formally, for a given set $\mathcal{V}$ of variables $\mathcal{V} = \{v_1, ..., v_n\}$ we call $v_i - v_j \leq c$ a potential constraint over $\mathcal{V}$. The variables as well as the constant must belong to the the numerical set $\bar{\mathbb{I}}$. $\bar{\mathbb{I}}$ can be either $\mathbb{R}, \mathbb{Z}$ or $\mathbb{Q}$ extended with the element $+\infty$. Then a set C of potential constraints over $\mathcal{V}$ can be represented by a $n \times n$ matrix m [Min01]:

$$m_{ij} \triangleq \begin{cases} c & \text{if } (v_j - v_i \leq c) \in C, \\ +\infty & \text{elsewhere.} \end{cases}$$

m is called Difference-Bound-Matrix.

To be able to encode all octagonal constraints ($\pm x \pm y \leq c$) it is necessary to have each variable twice in the DBM (m^+), in its positive and negative form. Thus, given a set of variables $\mathcal{V} = \{V_1, ..., V_n\}$ the set $\mathcal{V}' = \{V'_1, ..., V'_{2n}\}$ contains for each element V_i the positive form in V'_{2i-1} and the negative form in V'_{2i} [Min06]. Due to the expansion of the DBM to twice the original size, the former constraints have to be converted (c.f. Table 2.1) and afterwards they can be represented as a DBM.

Constraint			represented as					
$V_i - V_j$	$\leq c$	$(i \neq j)$	$V'_{2i-1} - V'_{2j-1}$	$\leq c$	and	$V'_{2j} - V'_{2i}$	$\leq c$
$V_i + V_j$	$\leq c$	$(i \neq j)$	$V'_{2i-1} - V'_{2j}$	$\leq c$	and	$V'_{2j-1} - V'_{2i}$	$\leq c$
$-V_i - V_j$	$\leq c$	$(i \neq j)$	$V'_{2i} - V'_{2j-1}$	$\leq c$	and	$V'_{2j} - V'_{2i-1}$	$\leq c$
V_i	$\leq c$		$V'_{2i-1} - V'_{2i}$	$\leq 2c$			
V_i	$\geq c$		$V'_{2i} - V'_{2i-1}$	$\leq -2c$			

Table 2.1: Constraints and their equivalent as octagonal constraint, taken from Miné [Min06]

Point-Wise Partial Order. A **Point-Wise Partial Order** $\trianglelefteq$ is induced by the $\leq$ order on the numerical set $\mathbb{I}$ on DBMs [Min01]: $m \trianglelefteq n \overset{\triangle}{\Leftrightarrow} \forall i, j, m_{ij} \leq n_{ij}$.

$\mathcal{V}$-domain. The **$\mathcal{V}$-domain** is any subset of $\mathcal{V} \mapsto \mathbb{I}$ satisfying the constraint

$$\forall i, j, v_i \in \mathcal{V}, v_j \in \mathcal{V} : v_j - v_i \leq m_{ij}.$$

The $\mathcal{V}$-domain is denoted by $\mathcal{D}(m)$. The **$\mathcal{V}^+$-domain** is the extension of the $\mathcal{V}$-domain, this is necessary due to the extension of the DBM. As the variables in $\mathcal{V}'$ are dependent ($v_{2i-1} = -v_{2i}$) the $\mathcal{V}^+$-domain is denoted by $\mathcal{D}^+(m^+)$ with:

$$\mathcal{D}^+(m^+) \triangleq \left\{(s_0, ..., s_{N-1}) \in \mathbb{I}^N | (s_0, -s_1, ..., s_{2N-1}, -s_{2N}) \in \mathcal{D}(m^+)\right\}$$

Because of $m \trianglelefteq n \Rightarrow \mathcal{D}^+(m) \subseteq \mathcal{D}^+(n)$ the ranges of the variables of m are a subset of the ranges of the variables of n when $m \trianglelefteq n$.

Operators and Transfer Functions. The **operators** and **transfer functions** on DBMs reach from equality and inclusion tests, over intersection and union to widening, assignments and constraint satisfaction tests. While most of these operators are defined quite intuitively, widening, assignments and the constraint tests need some further explanation. The idea of the widening $\triangledown$ is to remove the constraints in m^+ that are not stable by union in n^+:

$$[m^+ \nabla n^+]_{ij} \triangleq \begin{cases} m^+_{ij} & \text{if } n^+_{ij} \leq m^+_{ij}, \\ +\infty & \text{elsewhere.} \end{cases}$$

For assigning a linear arithmetic expression e to a variable $v_i \in \mathcal{V}^+$, what can be expressed as $v_i \leftarrow e(v_0, ..., v_{N-1})$, we write $m^+_{(v_i \leftarrow e)}$ for the DBM representing the set of possible values of $\mathcal{V}^+$. Except for some cases, where over-approximation is done, the assignment function is exact.
For the result of a successful constraint satisfiability test g on m, we write $m^+_{(g)}$ for the DBM representing the set of possible values of $\mathcal{V}^+$. It is exact in most cases, like the assignment function. All operators and transfer functions are described in more detail in the related literature [Min01].

Example for Octagonal Constraints and a DBM

The name of the octagon abstract domain refers to the amount of edges the shape features at most in the second dimension. The constraints bounding the octagon from Figure 2.4 could be:

- $X \geq 1$
- $X \leq 6$
- $Y \geq 6$
- $Y \leq 5$
- $Y - X \leq 3$
- $X + Y \leq 10$
- $X - Y \leq 4$
- $-X - Y \leq -3$

After converting these constraints to the form described in Table 2.1 they can be shown as a DBM. The matrix resulting from the constraints shown in Figure 2.4 part (III) and mentioned in the list above can be seen in the Table 2.2. Further information about the internal representation of constraints and assignments in the Octagon Abstract Domain can be found in the related literature [Min06].

2.5.2 The Octagon Abstract Domain Library

Based on the Octagon Abstract Domain introduced in the last section, the Octagon Abstract Domain Library[6] was implemented. It is written in C and also provides an OCaml interface.

[6] The current version is 0.9.10, it can be found here: `http://www.di.ens.fr/~mine/oct/`

	1	2	3	4
1	$+\infty$	-2	3	-3
2	12	$+\infty$	10	4
3	4	-3	$+\infty$	-2
4	10	3	10	$+\infty$

Table 2.2: Difference-Bound Matrice for the octagon from Figure 2.4

Library Configuration Options

The Octagon Abstract Domain Library has some options which can be configured at compile time. The most important one is the ability to choose the underlying numerical representation from integers, fractions, floats and the related GMP[7] and MPFR[8] types. This way one can chose between fast computation which is unsafe in case of overflows (integer representation) and slower, more memory consuming but also more precise computation with floats, or the arbitrary precision types. The choice of which numerical representation should be used depends on the program which should be analyzed. For most programs where only integers occur, the integer representation suffices, for programs where floats occur the float representation should be used.

Library Interface[9]

The library provides methods for creating octagons with a parameterized amount of variables (dimension) which are either unsatisfiable (bottom) or always satisfiable (top). Like an equality test, also tests on top and bottom exist. The available operators reach from the intersection of octagons to the widening of one octagon with another one. Regarding transfer functions, the library is able to forget the information which is stored about a variable, it can handle all the constraints introduced in the last section, and also handle variable assignments with both intervals and distinct values. Additionally the dimension of the octagon can be changed. Because of this, variables can be removed and added from and to the octagon. Furthermore both the

[7] GMP is a library for arbitrary precision arithmetic on integer, rational and floating-point numbers. C.f. `https://gmplib.org/`

[8] MPFR is a library for multiple-precision floating-point computations with correct rounding. C.f. `http://www.mpfr.org/`

[9] The complete documentation of the library's interface can be found here: `http://www.di.ens.fr/~mine/oct/current/doc/doc_oct.pdf`

constraints of the octagon and the interval for each variable can be printed out. Each of these operations creates a new octagon, while the old one is immutable.

2.5.3 APRON Library Architecture & Interface

In contrast to the Octagon Abstract Domain Library, the APRON[10] library aims at providing a generic interface to several abstract domains. The octagon abstract domain, an abstract domain for boxes and one for polyhedras are included.

Configuration Options

Like the Octagon Library, the APRON library is configured at compile time. Through several options in the Makefile, additional abstract domains and OCaml / Java bindings can be enabled.

Library Interface

The interface of the APRON library differs considerably from the Octagon libraries interface. It is more abstract and allows a simpler creation of variable assignments and constraints. As in the Octagon Abstract Domain Library, octagons can be joined, widened and intersected. Additionally the APRON library is able to differentiate between integer and float variables.

2.5.4 PAGAI

PAGAI [HMM12] is a freely available static analyzer written in C. It uses the octagon abstract domain implemented in the APRON library. Unlike CPAchecker PAGAI takes LLVM[11] bitcode as input. It is also not able to check the analyzed program on custom specifications but only on the standard C/C++ assert makro.

[10] The APRON library was also implemented by Antoine Miné, it can be found here: `http://apron.cri.ensmp.fr/library/`

[11] The LLVM Project is a modern compilation framework. Its intermediate representation is called bitcode. More information can be found here: `http://llvm.org/`

3 Octagon-Based Software Verification

In this chapter we define a CPA+ (c.f. Section 2.2) based on the octagon abstract domain. It will be called Octagon CPA. At first, an overview of the basic CPA+ parts is given. Later on some advanced strategies for improving the Octagon CPA are introduced.

3.1 The Octagon CPA

The formal requirements of a CPA+ were introduced in Section 2.2. The realization of these requirements, and so the definitions of all essential parts of the Octagon CPA are given in this section.

The Abstract Domain. The three pieces of the abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ are a set C of concrete states, a semi-lattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ and a concretization function $[\![\cdot]\!] : E \to 2^C$. The semi-lattices components are:

- an infinite set E of octagons (these are represented by DBMs),
- a top element $\top$, thus an octagon with no constraints or specific assignments (in the DBM each cell has the value $+\infty$),
- a bottom element $\bot$ which is an octagon that is, due to some constraints, not satisfiable at all,
- a preorder $\sqsubseteq \subseteq E \times E$ represented by the point-wise partial order $\trianglelefteq$ on DBMs, introduced in Section 2.5.1, for two abstract states e, e' we define $\sqsubseteq (e, e') = e \trianglelefteq e'$.
- and a total function $\sqcup : E \times E \to E$, available through the union operator $\cup$ on DBMs, for two abstract states e, e' we define $\sqcup(e, e') = e \cup e'$.

The concretization function $[\![\cdot]\!] : E \to 2^C$ is given through the $\mathcal{V}^+$-domain.

The Set Π of Precisions. The set Π is a power set of all program variables X, such that $\Pi = 2^X$. A single precision defines the variables which should be tracked during the analysis.

The Transfer Relation. In the transfer relation the assignment of linear expressions to variables and assumptions are handled. For assignments, the assignment transfer function $m^+_{v_i \leftarrow e}$ on DBMs (c.f. Section 2.5.1) is used. Equally to the assignment transfer function there is a transfer function $m^+_{(g)}$ for testing whether a certain constraint g holds. It is used for handling assumptions in the transfer relation.
Both, assignments and assumptions, depend on the precision. This means that assignments to untracked variables are ignored, and assignments from untracked variables to tracked variables result in forgetting all limitations and constraints of this variable (its interval is afterwards $[-\infty, +\infty]$). The same interval is used when any assumptions about untracked variables should be made.

The merge Operator. In CPAs there can be several different versions of the merge operator. For octagon-based CPAs three different operators are considered in this bachelor's thesis. These reach (related to an analysis) from very precise but slow to very imprecise but much faster. For working correctly, all octagon merge operators assume both octagons belong to the same program location. The differences between the three versions are explained in the next sections.

The merge^{sep} Operator. The merge^{sep} operator is the most precise merge operator and one of the standard merge operators. It never merges states, so there is no over-approximation caused by unifying two octagon states. For two abstract states e, e' and a precision π the operator is defined by $\text{merge}^{sep}(e, e', \pi) = e'$.

The merge^{join} Operator. The merge^{join} operator joins two abstract states using the total function $\sqcup : E \times E \to E$ defined in the abstract domain. It is also one of the standard merge operators. For two abstract states e, e' and a precision π the merge^{join} operator is defined by $\text{merge}^{join}(e, e', \pi) = e \sqcup e' = e \cup e'$. It is therefore not as precise as the merge^{sep} operator.

The $\mathsf{merge}^{widening}$ Operator. The $\mathsf{merge}^{widening}$ operator is the most imprecise merge operator. It over-approximates even more than the merge^{join} operator, due to using the widening transfer function $\triangledown$ of the DBMs instead of the union operator. For two abstract states e, e' and a precision π the $\mathsf{merge}^{widening}$ operator is defined by $\mathsf{merge}^{widening}(e, e', \pi) = e \triangledown e'$. On widening one octagon with another, each variable gets the maximal possible value range assigned, depending on all variable relations in these octagons, such that the least stable constraint by union is removed (c.f. Section 2.5.1).

In practice this means that loops are not unrolled. Instead the maximal possible state for each control flow edge (with call-stack) is computed at the second occurrence of the same control flow edge (the second iteration of a loop). All states for these locations, which will be computed afterwards (e.g. in the third iteration of the loop) are already covered by the widened state, and therefore they do not have to be considered once again in the abstract successor computation.

The Termination Check stop. The termination check stop is based on the preorder $\sqsubseteq$ of the abstract domain. It uses the point-wise partial order $\trianglelefteq$ on DBMs for inclusion testing. For two abstract states e, e' the termination check is defined by $\mathsf{stop}(e, e', \pi) = e \sqsubseteq e' = e \trianglelefteq e'$.

The Precision Adjustment Function prec. For the Octagon CPA precision adjustment is not used, thus the precision adjustment function prec can be defined for two abstract states e, e' and a set $R \subset 2^{E \times \Pi}$ as $\mathsf{prec}(e, \pi, R) = (e, \pi)$.

3.2 Influence of the merge Operator on the Analysis

In the introduction of the merge operators their preciseness and degree of over-approximation was briefly described. To further illustrate this, the example from Listing 2.1 is taken. It features a loop with a non-deterministic loop exit condition and an increasing variable inside the loop. Thus the analysis will, in contrast to the $\mathsf{merge}^{widening}$ operator, never stop when using the merge^{sep} or the merge^{join} operator. The Table 3.1 shows some measures and their values for each of the merge operators. They will be explained in the following paragraph. Afterwards these values will be compared and interpreted.

<table>
<tr><th></th><th>merge^{sep}</th><th>merge^{join}</th><th>merge^{widening}</th></tr>
</table>

(a) Results for the CFA in Figure 2.1

<table>
<tr><td>terminates</td><td colspan="2">no</td><td>yes</td></tr>
<tr><td>result</td><td colspan="2">/</td><td>true</td></tr>
<tr><td>size of the final set reached</td><td rowspan="2">$2 + 9 * \#LoopTraversals$</td><td colspan="2">11</td></tr>
<tr><td>amount of states created</td><td>$2 + 9 * 2 * \#LoopTraversals$</td><td>30</td></tr>
<tr><td>precision</td><td>single values for ticks in each state</td><td>slowly increasing interval for ticks in the state for each location</td><td>after the widening ticks is in the interval $[0, +\infty]$</td></tr>
</table>

(b) Results for the CFA in Figure 2.1 but with changed loop condition: $ticks < 100$

<table>
<tr><td>terminates</td><td colspan="3">yes</td></tr>
<tr><td>result</td><td colspan="3">true</td></tr>
<tr><td>size of the final set reached</td><td rowspan="2">902</td><td colspan="2">11</td></tr>
<tr><td>amount of states created</td><td>1802</td><td>30</td></tr>
<tr><td>precision</td><td>single values for ticks in each state</td><td>slowly increasing interval for ticks in the state for each location, finally $[0, 100]$</td><td>after the widening ticks is in the interval $[0, 100]$</td></tr>
</table>

Table 3.1: Results of an analysis with the Octagon CPA when using different merge operators

Measures and Their Computation. Besides self explanatory measures like termination, the result, or the precision, two other measures are interesting for comparing the merge operators. These are the size of the set reached and the overall amount of created states. The set reached has an impact on the amount of checks with the stop operator and also greatly influences the evaluation performance due to the amount of states which have to be kept. By contrast, the overall amount of created states includes not only the states which are in the set reached, but also all states which are created during the analysis. Both measures can be computed as following for the example CFA in Figure 2.1:
There are two locations before the loop head. The amount of all other reachable locations, inside the loop, as well as outside, is nine[1]. The merge^{sep} operator now considers both subtrees each time the loop head is reached (endless (a) or 100 times for (b)). As there is no new state created while merging, and instead both sets are kept in the set reached, this set, and the amount of created states are equal. With merge^{join} each of the nine locations has to be considered twice: once for the abstract successor computation, and once again for the state created in the merge step. This state then replaces both other states in the set reached. For this reason, for the merge^{join} the amount of created states is much higher (even infinite for the non-deterministic loop) than the size of the set reached. With $\text{merge}^{widening}$ the two subtrees of the loop head at location three have to be considered separately. The widening is done in the second traversal of the loop, starting with location three, two states are created (one for the $\text{merge}^{widening}$, and one for the abstract successor computation). Afterwards the left subtree is visited once again (6 states, for each a widening and an abstract successor) and the right subtree, namely the loop is also unrolled another time. On reaching the loop head the created state is equal to the one in the set reached, thus the stop operator stops the abstract successor computation at this point, and the overall amount of created states is 30.

Differences of the merge Operators. In Table 3.1 it can be seen that a $\text{merge}^{widening}$ is not only creating less states, it also over-approximates, such that instead of a distinct value for ticks, in (a) the interval $[0, +\infty]$ is saved. In case of a bounded loop (b) the interval is $[0, 100]$. While for an analysis using the Octagon CPA with the

[1] The locations 12, 13 and 14 can never be reached. This is because of the assumptions before. At first the variable $flag$ needs to be greater than zero in order to get to location 9. And at second, at location 9 the variable would have to be smaller or equal to zero in order to reach location 12. This is not possible.

merge^{sep} operator the size of the set reached is the same as the amount of states which are created, an analysis with the merge^{join} operator creates much more states (after the computation of the abstract successors, the merge function creates an additional new state which replaces both other states) as there are in the set reached. Both analyses do only terminate if the loop is bounded. The $\text{merge}^{widening}$ operator is not restricted by unbounded loops. Through the coarse over-approximation after the second iteration of the loop each of the created states is covering the states, that will get created in the third loop iteration. This is an advantage on the one hand, here the analysis terminates and even produces the correct result, but on the other hand if the error location would not be dependent on the variable $flag$ but on the variable *ticks* the result would probably be a false negative. The merge^{join} operator has the same disadvantage.

3.3 Advanced Algorithms for the Octagon CPA

The theory for the Octagon CPA introduced in the last section suffices to implement a working analysis. This analysis will then either have a huge state space or it will be very imprecise. In the following two sections two approaches are introduced, that reduce those disadvantages. On the one hand CEGAR will be applied to the Octagon CPA, and on the other hand the performance will be improved by combining different configurations of the Octagon CPA sequentially.

3.3.1 The Octagon CPA and CEGAR

One of the major problems of the Octagon CPA with a merge^{sep} operator is the exploding state space. Through CEGAR the state space can be kept as small as possible, because only the variables necessary to prove all occurring counterexamples infeasible are tracked. These variables are found with the interpolation of the explicit-value analysis. As the explicit-value analysis is in some cases not precise enough[2] to be able to prove a counterexample infeasible, a second feasibility check on error paths is done with an octagon-based analysis. If the counterexample can be proved infeasible there, the variables which have to be tracked additionally can be computed. By taking the difference of the current precision and the variables which are collected from all in the path occurring assumptions (or variables they

[2] In cases with non-deterministic or interval values for variables, octagons can assume interval bounds for the variable whereas the explicit analysis is not able to save any value at all.

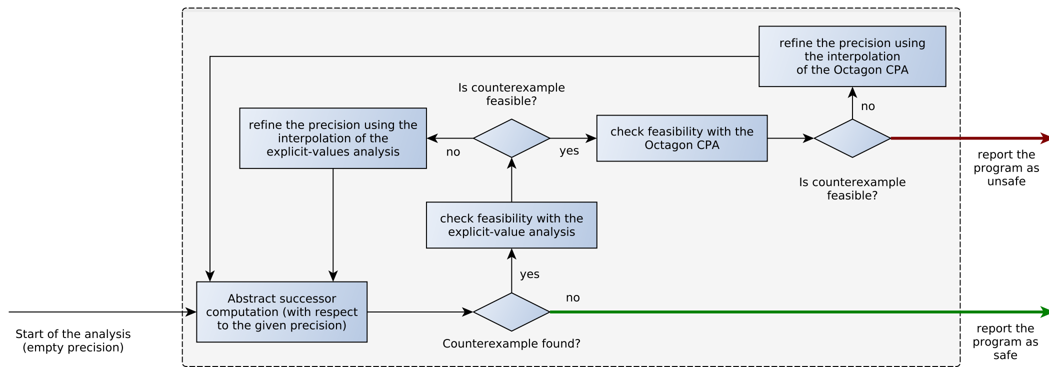

Figure 3.1: The Octagon CPA and CEGAR

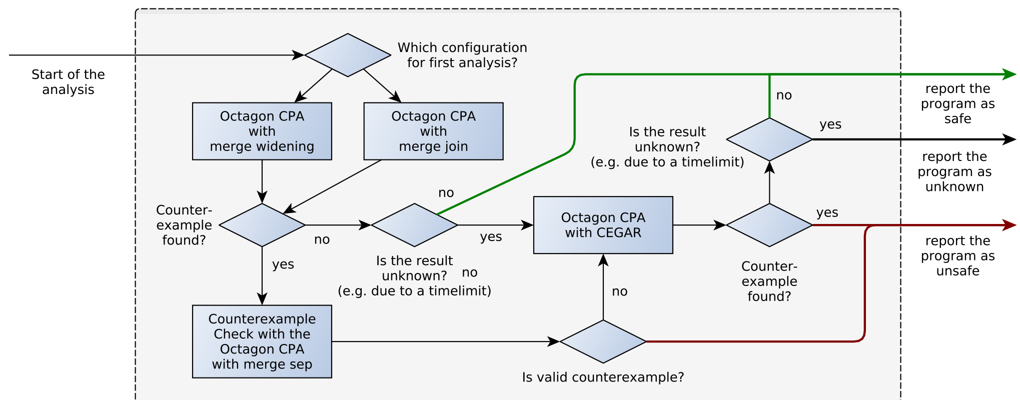

Figure 3.2: Sequential combination of different configurations of the Octagon CPA

depend on) we get the necessary precision increment. In this way, the efficiency of the explicit value interpolation is combined with the precision of the octagon-based analysis, thus the state space does not grow that much and more programs can be analyzed. The flow diagram in Figure 2.4 illustrates the usage of CEGAR with the Octagon CPA.

3.3.2 Sequential Combination of different Configurations of the Octagon CPA

In contrast to the last section where the improvement of analyses based on the Octagon CPA using a merge^{sep} operator was addressed, this approach aims at creating new analyses out of different Octagon CPA configurations. The restart algorithm (c.f. Section 2.3.4) provides an appropriate possibility to sequentially combine different analyses. It makes sense to use a fast configuration of the Octagon CPA (for example

one with a $\mathsf{merge}^{widening}$ operator) first (with the most precise Octagon CPA as counterexample check). And afterwards, if necessary, a more precise analysis using the merge^{sep} operator. By defining upper bounds for time or memory consumption, we can limit the run time of single analyses. When one of these limits is reached, the currently executed analysis is stopped, and *unknown* is reported as result. Figure 3.2 shows both combinations of configurations, that are considered in this bachelor's thesis.

4 Implementation of Octagon-Based CPAs

For the implementation of the Octagon CPA in CPACHECKER two octagon libraries were used. The Octagon Abstract Domain Library which exists in its current version 0.9.10 since April 2006 and the APRON library which is still maintained. While both libraries support the creation and manipulation of octagons, the main difference is their API. The changes in the programming interface are too huge to have only one CPA which can use both libraries as back-end. Instead two CPAs, one for each library were implemented. These implementations are described in detail in the following two sections.

4.1 A CPA Using the Octagon Abstract Domain Library

The Octagon Abstract Domain Library (OADL) is a library written in C. Through the Java Native Interface (JNI) this library will be used as back-end for the OADL CPA[1]. As mentioned in Section 2.5.2, the library can be compiled with different numerical types used as internal representation. This influences the performance of the library regarding speed and accuracy. Two library configurations were considered useful for this bachelor's thesis:

- The fastest and most memory-saving configuration with integers as internal representation will be used for analyzing integer related programs,
- and the library configuration with floats as numeric type for analyzing programs with floats.

[1] In the implementation all classes and packages are prefixed with "octagon" instead of "OADL", also the CPA is called OctagonCPA, the name was only changed in this thesis in order to emphasize the difference between the CPAs based on the APRON library and the Octagon Abstract Domain Library.

The choice which library should be used is made via a configuration option. Everything besides numerical variables, like pointers, structs, unions, arrays or strings is over-approximated, thus it is always assumed to have an unknown value. Cast expressions are ignored as well. The ability to represent relations between variables in the OADL CPA is one of the major advantages compared to the explicit-value analysis. In the following sections the OADL CPA is described in detail.

4.1.1 Architecture Overview

The OADL CPA has the same architecture as every other CPA, hence the general architecture is not described in detail. However there are some implementation features and enhancements for the CPA that are not intuitive. These are shown in the following sections.

The Octagon and NumArray Classes

As we are accessing the OADL via JNI the octagon object cannot be used directly in CPACHECKER. Thus for having the relation between an octagon in the OADL and an octagon in the CPA, the C pointer to the octagon is wrapped in an object of the class *Octagon*. Additionally the OADL has no memory management like it is known from Java, therefore we have to do another implementation trick in order to free octagons we do not need any more. When an octagon object is garbage collected in Java we do not get notified. One possibility to free the referring octagon in the OADL is by overwriting the finalize() method. Because this is deprecated (one cannot rely on the execution of finalize()) another approach using phantom references was implemented. Thus by creating an *Octagon*, a phantom reference is created additionally. This reference is then added to a queue when the corresponding Java object cannot be addressed any longer, and was garbage collected. At this point we do now free the octagon with the memory management functions of the OADL.

While integers or doubles can be easily used as parameters for functions called over the JNI, sometimes more complex parameters of different types are needed. The OADL requires for assignments an array of the library specific type num_t. Equally to the handling of octagons we use the same approach here. Instead of having the array as a parameter for the functions, only the pointer to the num_t array is exchanged between CPACHECKER and the OADL. This pointer is wrapped

in an object of the class *NumArray*. In contrast to the octagons, the num_t arrays are only needed in the scope of a function call, thus they can be allocated and freed directly before and after calling the function on the OADL. Therefore no additional code for the garbage collection is needed.

Coefficient Creation

The parameters of the functions *oct_assign_variable* and *oct_interv_assign_variable* are quite special. In the OADL for an assignment the index idx of the assigned variable $v_{idx} \in \mathcal{V}$, and a coefficient $coeff$ (either a single value or an interval) for each variable stored in the octagon plus one for a constant value are needed. The following formula represents the assignment:

$$v_{idx} \leftarrow \left(\sum_{i=0}^{N-1} v_i \cdot coeff_i \right) + coeff_N$$

The translation of an expression that should be assigned to a variable into such coefficients is handled in special classes. In order to being able to handle non-linear expressions at least partly, multiplication and division on the coefficients classes were implemented. An example for the additional feature is the non-linear assignment $x = x * y$ in the octagon o, which cannot be handled directly by the octagon abstract domain (and so the OADL). We are able to do parts of the multiplication during the coefficient creation process. By either substituting x with the value of x and using this value as coefficient for y or the other way round, we make the assignment linear. Hence, $x = x * y$ can be represented by $x \leftarrow \mathbf{valueOf}(x) * y$, and afterwards the OADL is able to handle this assignment.

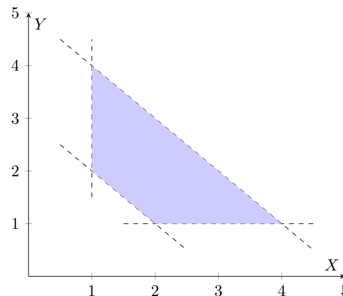

Figure 4.1: An example octagon

Although this is quite imprecise, it is the best approximation we can get with the octagon abstract domain. Figure 4.1 demonstrates this clearly. Despite the variables x and y are, each for itself, in the interval $[1,4]$, there are further constraints, such that $y <= 5 - x$ and $y >= 3 - x$. By resolving this non linear assignment through multiplying y with the value range of x which is $[1,4]$, the further constraints are lost, and the resulting value range for x is $[1,16]$ instead of the real range $[2,6.25]$.

4.1.2 Specific Configuration Options of the OADL CPA

In addition to the configuration options in Section 4.3.2, the OADL CPA provides further configurability regarding the handling of floats and the library usage:

- Via the option *cpa.octagon.octagonLibrary* one can choose which library should be used. It can either be set to *INT* or to *FLOAT*.
- The option *cpa.ocaton.handleFloats* toggles the float handling in the transfer relation. If turned off, float variables are ignored, if turned on, they are taken into consideration in the abstract successor computation. Enabling this option only makes sense while the version of the OADL compiled with floats as internal numerical representation is chosen. This option exists to be able to use the library compiled with floats, although only integer variables should be considered.

4.2 A CPA Using the APRON Library

The APRON library is like the OADL, written in C. Through the JNI this library will be used as back-end for the APRON CPA. For the APRON library there are also some options which can be set during compile time, for example additional abstract domains, or the JNI Wrapper files may be created and compiled. In contrast to the OADL the underlying numerical representation does not have to be changed in order to handle floats and integers, the APRON library can handle floats and integers at the same time. Because of that, the APRON CPA does not have the possibility to toggle the evaluation of floats. They are, as well as integers, always tracked depending on the used precision. Everything besides numerical variables, like pointers, structs, unions, arrays or strings is over-approximated. For this reason those variables are assumed to have an unknown value. Cast expressions for numeric variables are partly handled. The ability to represent relations between variables in

```
1 Texpr0BinNode right =
2     new Texpr0BinNode(
3           Texpr0BinNode.OP_ADD,
4           new Texpr0UnNode(
5                 Texpr0UnNode.OP_CAST, // several options for creating
6                 Texpr0Node.RTYPE_INT, // a cast that behaves like
7                 Texpr0Node.RDIR_ZERO, // original c code
8                 new Texpr0CstNode(new DoubleScalar(1.2))
9           ),
10          new Texpr0DimNode(1) // the variable b, referenced by its index
11    );

13 int assignedVarIndex = 0; // the variable a referenced by its index

15 state.assignCopy(
16    manager, // the manager, not important here, assumed to be a global
         variable
17    assignedVarIndex,
18    new Texpr0Intern(right), // the righthandside of the assignment,
         converted
19                             // to the internal type of the APRON
                                  library
20    null
21 );
```

Listing 4.1: Example to show the use of tree expressions for assignment creation in the APRON library

the APRON CPA is one of the major advantages compared to the Value Analysis CPA. Unfortunately there was not enough time to finish the implementation of this CPA. There are some bugs in the usage of the APRON library that make the APRON CPA unusable for larger programs[2]. This will be explained in the evaluation section. In the following sections the APRON CPA is described in detail.

4.2.1 Architecture Overview

The APRON CPA has the same architecture as every other CPA, so the general architecture is not described in detail. However there are some implementation features and enhancements for the CPA that are not intuitive. Through the object-

[2] The expectation of this CPA was to be at least as precise as the OADL CPA. Both libraries and CPAs are using octagons, thus their ability to prove constraints and handle assignments are equal. However the APRON library handles some inputs for functions in other ways, than we expect them to be handled, and there is no proper documentation about these border cases.

oriented implementation of the JNI wrapper the APRON library greatly takes away programming work from the CPA developer. There is for example no need for coefficient classes, like they are used in the OADL CPA. The parameters for the assignment and constraint creation functions are tree and linear expressions. These are organized like normal c expressions. The assignment:

$$a = ((int)1.2) + b;$$

could for example be converted to the function calls to the APRON library in Listing 4.1 (assuming *a* and *b* are variables 0 and 1 in the APRON state *state*). This example also shows another big advantage of the APRON library compared to the OADL, the ability of handling floats and integers at the same time, and casts. For the OADL the coefficients classes were introduced in order to create the parameters of the library functions. These classes also contain logic in order to handle multiplication and division. With the APRON library this logic is already implemented inside the library. Through transformations, equal to those implemented in the coefficients classes, the APRON library is able to handle non-linear expressions, too. These are also over-approximated, because octagons are not able to handle polynomial or exponential assignments and constraints. The following paragraphs briefly describe each class, that is important for the APRON CPA.

The ApronState. Each ApronState is a wrapper around an APRON object. The APRON object (called *Abstract0* in the implementation) is like in the OADL CPA holding a pointer to the object of the APRON library. As the APRON library also includes the JNI part, it will not be described in more detail. Besides the APRON object, the ApronState also keeps track of the mapping from variable names to their referring ids in the APRON object. These ids are more complex than the ones of the OADL state. In an ApronState one has to differentiate between float and integer variables. Both variable types have their own dimensions. However if one wants to add a variable the overall index is needed. For integers this is the index in the integer domain, for floats this is size of the dimension of the integer domain plus the index of the float variable in the float domain.

4.2.2 Specific Configuration Options for the APRON CPA

In addition to the configuration options in Section 4.3.2, the APRON CPA provides the choice which abstract domain should be used[3]. It is made via the option *cpa.apron.domain* and can either be set to

- *BOX*, an abstract domain which uses boxes,
- or *POLKA*, an abstract domain which uses and manipulates convex polyhedra,
- or *POLKA_STRICT*, an subtype of the POLKA domain which is able to represent strict inequalities,
- or *POLKA_EQ* an abstract domain which can manipulate linear equalities,
- or to *OCTAGON*, which is the abstract domain introduced during this bachelor's thesis, and which will therefore be used for the analyses with the APRON CPA.

4.3 Common Parts of the OADL CPA and the APRON CPA

This section mentions all parts of the OADL CPA and the APRON CPA which are equal up to their class names. As both CPAs have the same goal, and both CPAs use octagons, this is the case for many higher level parts of the CPAs. Thus everything that is not related to the underlying libraries, such as most configuration options, the precision and also CEGAR is described here.

Two Implementations of Precisions. For the OADL CPA and the APRON CPA two precisions were implemented. A static precision which is already "full" in the beginning and thus tracks all variables that can be tracked at every point of the analysis is the first implementation. It is used for all configurations besides the ones using CEGAR. The second precision was explicitly created for the use with CEGAR. It is refineable (variables that should be tracked additionally, can be added to the precision) and starts "empty", hence no variables are tracked at the beginning of the analysis.

[3] The information about the different options was found in the Java documentation of the JNI wrapper.

4.3.1 The CEGAR Implementation

For the implementation of CEGAR in CPAchecker, a refiner which finds the variables that have to be tracked in order to refute a found error path is needed. The interpolation step is quite difficult and costly. For that reason implementing the complete refinement would go beyond the scope of this bachelor's thesis. Hence the explicit-value analysis, which is relatively similar to an analysis with the OADL and APRON CPA, is used for computing the interpolants. Through this step we get interpolants[4] precise enough for many programs. However this step is not applicable to all kinds of analyzed programs or at least not to every part of the analyzed program.

In the example program from Listing 2.1 the $VERIFIER_error()$ call[5] is unreachable. But as the explicit-value analysis cannot handle intervals, the path feasibility check with the explicit-value analysis does not yield the expected result. In contrast, it reports a feasible counter-example where only the variable $flag$ would have to be tracked to prove that the complete program satisfies the specification.
In order to compensate this weakness of the explicit-value analysis, the refiners of the OADL CPA and the APRON CPA have a backup feasibility checker which is made with octagons. It is used when the explicit-analysis cannot successfully refute a found counterexample. Their feasibility checker uses a precision which is initially full to check the found error-path on feasibility. If the counterexample is still feasible with the feasibility checker of the OADL and APRON CPA, the verification process ends, and the counterexample violating the specification is returned. Otherwise their refiners compute their own interpolant in a very basic way:

1. The feasible prefix of the infeasible counterexample is computed with the feasibility checker.
2. Then the variables that occur in the assumptions (and their dependencies, e.g. assignments of other variables to this variables) of the feasible prefix get computed and are added to a set.
3. The difference between the previously tracked variables and the variables in the computed set represent the final precision increment.

[4] variables which have to be added to the precision in order to refute the found counterexample
[5] By the if conditions the variable $flag$ is limited to certain interval bounds. At first, if $flag$ is in the interval from $[-\infty, 0]$ the program quits, thus $flag$ has to be in the interval $]0, \infty]$ when the program reaches the second if condition. This condition is then fulfilled because $]0, \infty] > 0$, and the program has to quit here in every case.

With the found precision increment, finding the same error path in the next CEGAR iteration can be avoided. This approach is quite coarse and adds often variables to the precision which would not be needed in order to refute the counterexample. For that reason, this kind of interpolation is only used if the explicit-value analysis fails to refute a counterexample. If the interpolation is done once with octagons, the following CEGAR iterations can only be done with octagons, as well.

4.3.2 Configuration Options

The configurability of CPACHECKER is a big advantage. In one CPA several options can be defined which change the behavior of the analysis. For analyses with the OADL CPA and the APRON CPA there are some options which need to be set in order to make it work properly and others which can be chosen freely. Because of that, a "basic configuration" that is used by every other configuration using the OADL CPA or the APRON CPA, can be created.

Basic Configuration Options. Besides declaring the LocationCPA, the Callstack-CPA, the FunctionPointerCPA and the OADL CPA / APRON CPA as components of the CompositeCPA only one option has to be enabled for a basic analysis. This option is named *cfa.moveDeclarationsToFunctionStart*. By enabling it, all declarations are moved to the beginning of each function. This makes dimension changes in the OADL easier and less costly[6]. These dimension changes are always necessary when adding variables or when comparing, unifying or widening states with different amounts of variables.

Configuration Options for Using CEGAR. In order to use the CEGAR algorithm, the previously mentioned configuration options have to be extended. At first CEGAR is enabled by setting the option *analysis.algorithm.CEGAR* to *true*. Then in the *cegar.refiner* option we either specify *cpa.octagon.refiner.OctagonDelegatingRefiner* or *cpa.apron.refiner.ApronDelegatingRefiner* as its value. Finally the initial precision type of the CPA has to be a refineable precision. This is realized by setting the option *cpa.octagon.initialPrecisionType* or *cpa.apron.initialPrecisionType* to *REFINEABLE_EMPTY*.

[6] Inserting or deleting new dimensions at the highest index of the DBM is less costly than for example in the middle, as no permutations have to be done inside the DBM.

Configuration Options for Using The Restart Algorithm. The restart algorithm is the sequential combination of different analyses. Whenever an analysis fails to compute a proper result, the next analysis is used until the last specified analysis is reached. In the option *restartAlgorithm.configFiles* all configurations of analyses are mentioned in the order they should be used during the restart analysis.

Additional Configuration Options In addition to the already mentioned configuration options, the OADL CPA and the APRON CPA provide further configurability regarding the merge operator, and a time limit for the feasibility check while using CEGAR. All configuration options are named equally for both CPAs, the part **domain** should be replaced by **octagon** for the OADL CPA and **apron** for the APRON CPA:

- The merge operator, *cpa.***domain***.mergeop.type* can be chosen among *SEP*, *JOIN* and *WIDENING*. The default configuration uses the merge^{sep} operator.
- By enabling the option *cpa.***domain***.mergeop.onlyMergeAtLoopHeads* (it is disabled per default), one can further influence the behavior of the merge^{join} and $\mathsf{merge}^{widening}$ operators. By only merging the states which occur at a loop head[7] the analysis becomes more precise. However, every time the loop head is reached, the join or widening is done. So on each iteration of a loop the information in the states becomes more abstract and therefore the analysis is less precise than an analysis with the merge^{sep} operator.
- The option *cpa.***domain***.refiner.timeForOctagonFeasibilityCheck* allows to adjust the time limit for the feasibility check made with octagons while using CEGAR. One can set the maximal time in seconds the check may last. Zero means there is no limit, which is the default behavior.

[7] A loop head is the entry point of each loop. For the CFA in Figure2.1 the location three is a loop head.

4.4 Comparison of the OADL CPA and the APRON CPA Regarding the Programming Effort

In the sections about the implementation of the OADL CPA and the APRON CPA some similarities and differences have already been mentioned. The advantages and disadvantages of both CPAs and libraries will now be regarded in a more detailed way from the view of a programmer.

Support. In contrast to the OADL, which was developed from 2000 to 2006[8], the APRON library is still maintained. On the one hand this is an argument against using the OADL, but on the other hand, the requirements and the scope of the octagon abstract domain did not change. Thus both libraries can be equally used without having any disadvantages due to missing support (despite recently found bugs).

Abstract Domains. The OADL is an an implementation of the octagon abstract domain. The API of the OADL provides all kinds of functions for creating and changing octagons. The APRON library is not only aimed at providing an API to an implementation of the octagon abstract domain, it also supports other abstract domains, like boxes or polyhedra. Hence there is the need for a larger interface. This interface is divided into a special part for each abstract domain and a generic part for the whole library. As the scope of this bachelor's thesis is the theory and implementation of octagon-based CPAs, both libraries are appropriate.

Programming. When it comes to the implementation of octagon-based CPAs with both libraries the important differences can be seen. At first the OADL has especially for the assignment functions a relatively complicated signature. A great part of the OADL CPA is therefore the conversion of expressions to a form that the OADL can handle. This includes some optimizations which enable us to handle at least some non-linear expressions, what would not be possible otherwise. Second, the OADL is only able to handle either floats or integers at time. Compared to the APRON CPA this is quite inconvenient.

[8] More details can be found in the documentation of the library: `http://www.di.ens.fr/~mine/oct/current/doc/doc_oct.pdf`

The APRON library has a complex system of expressions, such that casts, binary expressions and unary expressions with arbitrary nesting can be represented. This works in a similar way as the representation in C code. This simplifies the creation of the parameters needed for the transfer functions of the APRON library. Additionally to the omission of the coefficient creation, the APRON library handles all statements which are too complex for a certain domain itself. For that reason the transformations on coefficients in the OADL CPA, that were made in order to be able to handle some non-linear assignments, are also not needed. Furthermore APRON is able to handle floats and integers at once. This is on the one hand an advantage, but it also leads to the disadvantages of APRON. For floats and integers there are no implicit casts as they exist in C. By assigning a float value which has decimal places to an integer variable the returned APRON object is always $\perp$. This is quite unintuitive. Throwing an exception or at least any kind of warning would be better. These $\perp$ states also occur if other parts of assignments fail and the same applies to constraint additions. Overall the OADL CPA was more implementation and theoretical work compared to the APRON CPA, the APRON CPA in contrast is harder to debug and has unintuitive return values for functions which are called with inappropriate parameters.

5 Evaluation

In this chapter the OADL CPA and the APRON CPA will be evaluated in terms of their performance with different kinds of programs. Also several configurations of the CPAs are taken into consideration. The evaluation is not limited to analyses in CPACHECKER, another analyzer, PAGAI, which uses octagons as well, is used for comparison purposes.

5.1 Benchmark Programs

For the evaluation and comparison many kinds of programs will be used. From the benchmark suite of the International Competition on Software Verification (SV-COMP), the appropriate categories were chosen. And we have some custom benchmark files, for illustrating the abilities of the octagon abstract domain on certain kinds of problems. Additionally there are some benchmarks that require the handling of floats, which are taken into consideration as well.

The SV-COMP Benchmark Set. The benchmark set of the SV-COMP features many categories which cover different program types[1]. Some categories, namely *Memory Safety*, *Concurrency*, *Recursive*, *Heap Manipulation* and *Bit Vectors* use features like recursion, concurrency, bitvectors or pointers. As our configurations do not support these features those categories are excluded from the evaluation. The benchmarks of the SV-COMP that are applicable to our configurations are *Control Flow and Integer Variables*[2], *Device Drivers Linux 64*, *Sequentialized Concurrent Programs*[3] and *Simple*. These categories do not include pointers and are therefore considered

[1] A more detailed description of the certain categories can be found here: `http://sv-comp.sosy-lab.org/2014/benchmarks.php`

[2] For the *eca* files ranging from *Problem10_** to *Problem19_** the verdict is currently unclear. Hence these files are excluded from the evaluation.

[3] From the *Sequentialized Concurrent Programs* set only the files in the *systemc* folder are taken for the evaluation. The other files rely on function pointers and arrays. The octagon-based CPAs cannot handle either of them.

meaningful for the following benchmarks. All programs in these categories do rely on floating-point variables. Because of the variability of the programs in this benchmark, a good overview over the performance of the OADL CPA and the APRON CPA analyzing "real life" problems, is given. Overall 2310 files are included in this set.

CBMC Benchmarks for Float Analyses. For showing the ability to analyze programs with floating-point arithmetic, some artificial benchmarks from the regression testing suite of the analysis tool CBMC and a special suite called *CDFPL* are used[4]. Unfortunately those programs make use of non-linear expressions in assignments, so they do not perfectly match the abilities of the octagon abstract domain. Due to the lack of other, non-artificial, programs these benchmarks can only give an idea of the performance of the OADL CPA and the APRON CPA when real world problems should be analyzed.

Custom Programs. To compensate the gaps regarding certain kinds of programs some artificial benchmarks were created. For example, there is a program that relies on float variables. Other benchmarks can only be proved correct by considering relations between variables. These programs will be evaluated in an other way than the SV-COMP benchmark set. They should show certain abilities of the analyses and not make an overall statement on the performance of OADL CPA and the APRON CPA.

5.2 Configurations

For the evaluation we used several configurations of the OADL CPA. They are briefly described here:

oadlSep This is the basic configuration. It uses all default values for the implemented configurations options, so the merge operator is merge^{sep}.

oadlJoin In this configuration the merge operator is changed to merge^{join}.

[4] More information about CBMC can be found here:
http://www.cs.cmu.edu/~modelcheck/cbmc/
http://svn.cprover.org/svn/cbmc/trunk/regression/cbmc/
http://www.cprover.org/cdfpl/

oadlJoin-cex This is the **oadlJoin** configuration with counterexample check. If a counterexample occurs it is rechecked with the **oadlSep** configuration.

oadlJoin-LH This is the **oadlJoin-cex** configuration with a changed merge strategy. Instead of merging when possible, this configuration merges only states that are at loop heads.

oadlWidening In this configuration the merge operator is changed to $\text{merge}^{widening}$.

oadlWidening-cex This is the **oadlWidening** configuration with counterexample check. If a counterexample occurs it is rechecked with the **oadlSep** configuration.

oadlWidening-LH This is the **oadlWidening-cex** configuration with a changed merge strategy. Instead of merging when possible this configuration merges only states that are at loop heads.

oadl-refiner This is the basic **oadlSep** configuration which uses CEGAR additionally.

oadl-seq-J-R In this configuration the **oadlJoin-cex** configuration and the **oadl-refiner** configuration are sequentially combined. At first the **oadlJoin-cex** configuration is used for analyzing the program. If then a counterexample was reported which can be proved infeasible with the counterexample check or if the analysis is ended due to a time or memory limit, the second analysis with the **oadl-refiner** configuration is started. The first configuration is limited to 250 seconds time for the analysis, and the subsequent configuration gets all the rest.

oadl-seq-W-R In this configuration the **oadlWidening-cex** configuration and the **oadl-refiner** configuration are sequentially combined. By appending either **-100** or **-250** we express the run time limit for the widening configuration. At first the **oadlWidening-cex** configuration is used for analyzing the program. If then a counterexample was reported which can be proved infeasible with the counterexample check or if the analysis is ended due to a time or memory limit, the second analysis with the **oadl-refiner** configuration is started.

-float is appended to the configuration name to show that the float handling is enabled during the analysis. For the APRON CPA the basic configuration **apronSep**

with a merge^{sep} operator is used. Furthermore two configurations of the explicit-value analysis are used for comparison purposes. These are an analysis without CEGAR and without final counterexample check[5] (**eva-basic**), and an analysis with CEGAR but also without final counterexample check (**eva-refiner**). From the predicate analysis only the standard configuration (**predicate**) was considered.

5.3 Evaluation Environment

The evaluation was performed on machines with a 2.6 *GHz* Octa Core CPU (Intel Xeon E5-2650 v2) and 128 *GB* of RAM. The operating system is an Ubuntu 12.04 (64-bit) with a Linux 3.13.0-30 kernel. For the Java support OpenJDK 1.7 is used. The CPACHECKER revision for the evaluation is 12883[6]. Each single verification run was limited to 32 *GB* of RAM and to 500 seconds of run-time. The Java heap was set to 2 *GB* for analysis with the OADL and APRON CPA, to 29 *GB* for the explicit-value analysis and to 25 *GB* for the predicate analysis. PAGAI[7] was also run with 32 *GB* of RAM. For each verification run the overall amount of CPU time[8] and memory usage is measured. In all tables time and memory consumption will be given in hours and megabytes with two significant digits unless it is specified in another way. For the conversion of bytes we use SI units. This means that 1 *MB* is composed of 1000 *KB* not 1024 *KB*.

5.4 Performance Evaluation of the OADL CPA and the APRON CPA

In this section both implementations of the Octagon CPA that were introduced in this bachelor's thesis, are evaluated. Several measures, such as the number of

[5] The explicit-value analysis uses by default CBMC to recheck found counterexamples. In order to make it better comparable, this check is deactivated.
[6] Some minor changes that only effect the evaluation of the floating-point benchmarks were made in revision 12904 in the branch *octagon-bathesis*. Thus the floating-point benchmarks were performed on this CPACHECKER version.
[7] PAGAI can be downloaded as a statically linked binary here:
`http://pagai.forge.imag.fr/`
However, the downloadable version has the release number 14-04-07, but the version used here was especially compiled for us (14-04-09), and does not match the version which is on this website.
[8] This time measure refers to the CPU time of the whole verification run.

created states, the time and memory consumption of the analyses, and the amount of successfully analyzed programs are taken into consideration.

5.4.1 Evaluation of Integer-Related Programs

The integer-related programs from the International Competition on Software Verification build the largest part of the applied benchmark set. The variety of these programs is a big advantage compared to the small amount of floating-point related benchmarks. In this section these programs will be used to compare different configurations of the OADL CPA regarding their strengths and weaknesses. Unfortunately the APRON CPA contains bugs (c.f. Section 5.5) that make the analysis of large-scale problems, like they are included in the integer benchmark set, impossible. For that reason a comparison between these CPAs is not meaningful.

Overview. In Table 5.1 one can see the aggregated results of all integer-related benchmarks. Besides the **predicate** analysis, which is able to prove the most programs correctly, the **oadl-refiner** configuration is the third best single-analysis configuration after **eva-refiner** regarding the amount of successfully proved programs. It is able to successfully analyze about 200 to 300 programs more than the other single-analysis configurations using the OADL CPA. Compared to the **eva-refiner** configuration, eleven programs less could be verified successfully, but for twelve programs less a wrong result was returned. Big differences in the performance can be seen between configurations using differing merge operators. When it comes to the sequential combination of analyses, **oadl-seq-W-R-100** and **oadl-seq-W-R-250** achieved better results than the single-analysis configurations. The given time in the table refers to the overall amount of used computation time (including the time of verification runs that did not terminate or yield a wrong result). The bad result of the **apronSep** configuration is caused by the remaining bugs in the APRON CPA. All results are discussed in the following paragraphs.

Comparison of Single-Analysis Configurations. By taking a closer look at the single-analysis configurations we can determine three groups which are interesting for a further evaluation.

- The comparison of **oadlSep**, **eva-basic**, **oadl-refiner** and **eva-refiner** is worthwhile, as these configurations use the same merge strategy and also the same refiner. This illustrates the differences in the abstract domains in much detail.

Configuration	#correct	#false alarms	#false proves	Time (*h*)
Analyses with the OADL CPA and the APRON CPA				
apronSep	703	1162	1	46
oadlSep	1506	57	0	90
oadlJoin	1577	188	0	61
oadlJoin-cex	1433	35	0	77
oadlJoin-LH	1247	20	0	120
oadlWidening	1714	276	0	40
oadlWidening-cex	1507	19	0	53
oadlWidening-LH	1352	23	0	95
oadl-refiner	1801	50	0	52
oadl-seq-J-R	1549	37	0	83
oadl-seq-W-R-100	1870	51	0	50
oadl-seq-W-R-250	1842	51	0	56
Explicit-Value and Predicate Analyses				
eva-basic	1747	63	0	68
eva-refiner	1812	62	0	50
predicate	1994	11	1	33

Table 5.1: Overall Performance of the Analyses on the SV-COMP Benchmarks

- The comparison of **oadlSep**, **oadlJoin** and **oadlWidening**, allows a deeper insight into the differences of the merge operators.
- The comparison of **oadlSep**,**oadlJoin** and **oadlWidening** is interesting, because the latter use the **oadlSep** configuration as counterexample check. Therefore these analyses do already profit of a more precise second configuration.

In the following paragraphs, the results of the before-mentioned configurations will be discussed.

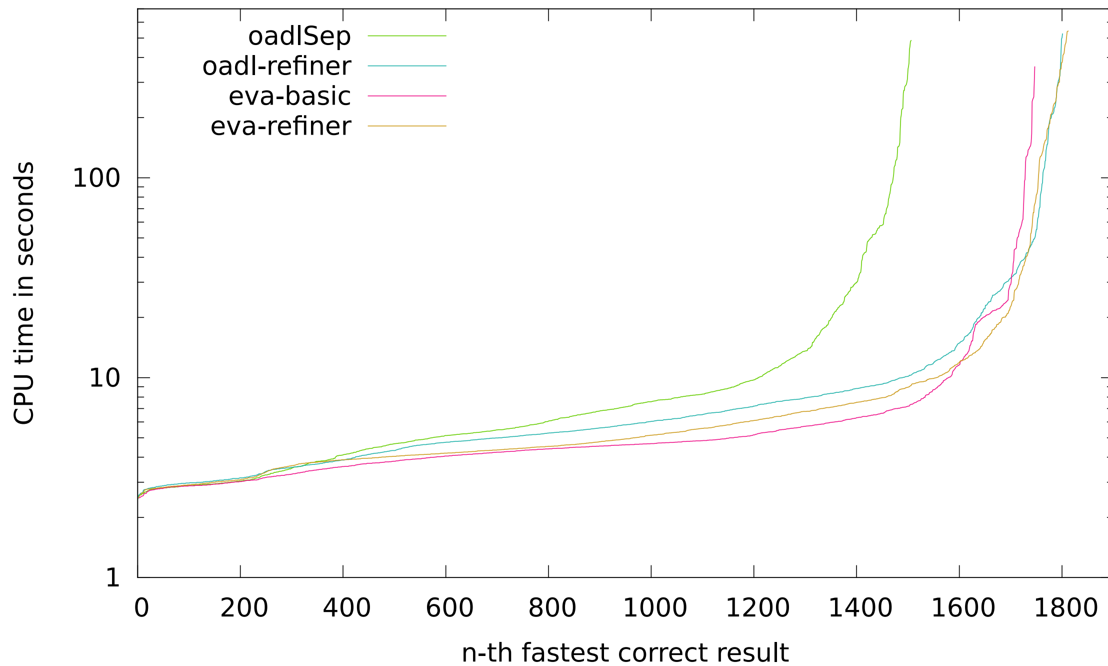

Figure 5.1: A plot of quantile functions of the configurations **oadlSep**, **oadl-refiner**, **eva-basic** and **eva-refiner**

Comparison of oadlSep, eva-basic, oadl-refiner and eva-refiner. When only regarding the correctly proved programs, one can see that the OADL CPA profits much more of CEGAR than the explicit-value analysis does (c.f. Figure 5.1). While the **eva-refiner** configuration is able to successfully analyze 65 programs more than the **eva-basic** configuration, for the OADL CPA the difference is larger. Instead of 1507 programs with **oadlSep**, 1801 programs can be analyzed successfully (c.f. Table 5.1). This is an improvement by approximately 20%. Table 5.2 gives some indications for that. At first, the average amount of states in the sets reached for analyses with the OADL CPA is much lower than with the explicit-value analysis.

Configuration	∅ #reached states	∅ #refinements	∅ memory (MB)
oadlSep	15000	–	1200
oadl-refiner	11000	6.0	630
eva-basic	42000	–	320
eva-refiner	16000	14	560

Table 5.2: Average Number of refinements, states in the set reached and memory consumption for certain configurations, only successfully-analyzed verification tasks are considered

The **oadlSep** configuration creates even less states than the **eva-refiner** configuration. The explanation for that behavior are the relations between variables that limit the amount of possible states. By using CEGAR (**oadl-refiner**) the state space is reduced once more. Also when looking at the average memory consumption per verification task, savings of over 50% can be seen. For the **eva-refiner** configuration in contrast the memory consumption increases while the state space is smaller than without using CEGAR. Another advantage of the availability of linear constraints is the smaller amount of necessary refinements.

Comparison of oadlSep, oadlJoin and oadlWidening. The analysis results of these configurations differ greatly. The amount of reached states and the memory consumption decreases (c.f. Table 5.3) with the precision of the analysis. This happens due to reaching error locations earlier than a more precise analysis would reach them. As no counterexample check is done at the end of the analyses, the amount of successfully proved programs is not a meaningful measure for these configurations. Although the report of an unsafe program with **oadlJoin** or **oadlWidening** may be correct, the counterexample found will in most cases not be a valid one. In this case a better measure for the performance is the number of created successors. While the **oadlSep** configuration creates the fewest successors of all configurations, **oadlJoin** creates the most. Due to the small amount of abstraction which is added when joining states, the **oadlJoin** configuration finds many new abstract states that are not covered by those that exist at this point. Additionally through the over-approximation many assumptions are considered satisfiable even if they are not. The **oadlWidening** shows a better approach for the merge than simply joining two states. Due to the widening the over-approximation is coarser than with **oadlJoin**.

Configuration	#reached states	#successors	∅ memory (MB)
oadlSep	109592186	113575288	5700
oadlJoin	53446165	336858179	4400
oadlWidening	25248833	190570792	2000
oadlJoin-cex	57508907	356710324	5700
oadlWidening-cex	24585622	210071521	3300
oadlJoin-LH	112254046	118817033	7300
oadlWidening-LH	105635978	260048702	7200

Table 5.3: Number of created states, states in the set reached and average memory consumption for certain configurations

This makes it the fastest and also most memory saving configuration among this three. By adding more over-approximation, newly created states are covered more often by already existing states, hence they do not have to be considered once again in the abstract successor computation. However more states which should not be reachable can be reached, and because of this more false alarms, compared to the **oadlJoin** configuration, are reported (c.f. Table 5.1).

Comparison of oadlSep, oadlJoin-cex and oadlWidening-cex. The counterexample check with the **oadlSep** configuration is a first approach on improving the imprecise **oadlJoin** and **oadlWidening** analyses. According to Table 5.1 this works quite well. The false alarms reported by the analyses can be reduced to a minimum. So both configurations, **oadlJoin-cex** and **oadlWidening-cex** are at least as precise (only with regard to false alarms) as the **oadlSep** configuration. When it comes to the time consumed, the counterexample configurations are also better, compared to analyses with **oadlSep**. While with the merge^{join} operator only about 10 hours of computation time can be saved, the $\text{merge}^{widening}$ saves 35 hours, over 30% of the time of merge^{sep}.

The Effect of Only Merging at Loop Heads. By only joining or widening states at loop heads the configurations are intended to achieve a higher precision inside of certain loop iterations. However, merging only at loop heads is counterproductive.

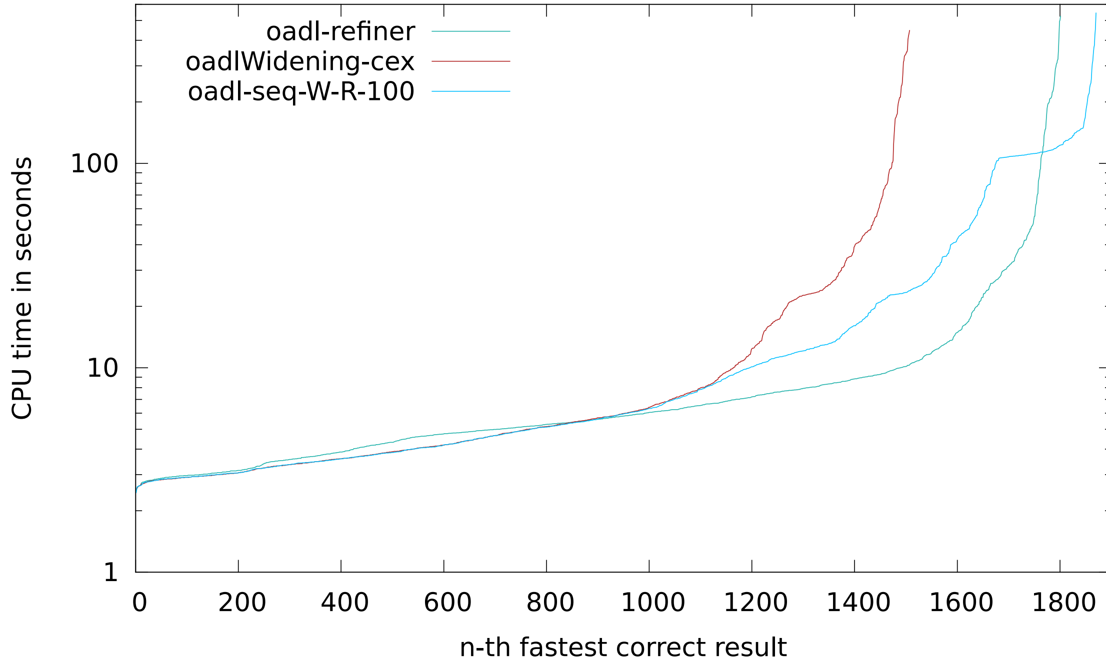

Figure 5.2: A plot of quantile functions of the configurations **oadlWidening-cex**, **oadl-refiner** and **oadl-seq-W-R-100**

While the initial state for each loop iteration becomes more and more abstract, the states which are created inside the loop are not merged. This leads to much higher amounts of created successors and reached states as usual for configurations which do a merge join or widening (c.f. Table 5.3). The precision of the analysis could not be increased, as the over-approximation at loop heads is too coarse. Because of that, these two configurations are considered the worst configurations that were tested.

Comparison of the Sequential Combinations of Analyses. When creating a sequential combination of analyses, the goal is to find analyses that complement each other. As one goal of this thesis is the evaluation of the Octagon CPAs that were implemented, only sequential combinations of these CPAs were considered. Table 5.1 gives an overview over all single-analysis configurations and their abilities. As expected of the results for the **oadlJoin-cex** and **oadlWidening-cex** configurations, the combinations with widening were clearly better. The optimization with using a smaller time limit of 100 seconds for the first analysis (instead of splitting up the time into two equal parts of 250 seconds), increases the correctly analyzed programs from 1842 to 1870. This can be explained by the results of the **oadlWidening-cex** configuration. The results are either computed fast or they tend to be unknown due

Configuration	#correct	#false alarms	#false proves	Time (h)
oadlSep	632	45	0	15
oadl-seq-W-R-100	619	35	0	28
PAGAI	322	0	11	30

Table 5.4: Overall performance of the analyses on the SV-COMP benchmarks

to the time limit. The reason for exceeding the time limit is that some conditions cannot be widened. In such cases, the **oadl-refiner** configuration is more likely to return a proper result. So the configuration **oadl-seq-W-R-100** is considered being the best among the ones using the OADL CPA. Furthermore, it is able to correctly analyze 58 programs more than the explicit-value analysis while they take about the same time. Additionally eleven programs less are analyzed wrongly. Figure 5.2 shows the components and the combinations results. The x-axis denotes the number of the program (they are sorted ascending after run time) and the y-axis the amount of time, that each verification run took.

Comparison of the OADL CPA to PAGAI PAGAI is a static analyzer which can use the octagon abstract domain for its verification process. Due to the lack of further benchmarks that are prepared in a way that PAGAI is able to handle them, the SV-COMP set had to be reduced to the categories *Simple* and *Control-Flow Integer*. The total number of benchmark programs used is therefore 819. While the **oadl-seq-W-R-100** and **oadlSep** configurations are able to analyze over 600 programs correctly, PAGAI is only able to successfully analyze 322 files (c.f. Table 5.4). Furthermore it produces eleven false proves. Regarding the amount of CPU time, the **oadlSep** configuration is approximately twice as fast as PAGAI.

5.4.2 Evaluation of Float-Related Programs

The floating-point arithmetic benchmarks cover only a small range of possible programs. All benchmarks in this set are artificial, therefore the results do not have the same significance as the benchmarks from the SV-COMP set.

CBMC Regression and *CDFPL* Benchmark Suite. Because of the small programs, analyses with CEGAR or sequential combinations of other configurations

Configuration	#correct	#false alarms	#false proves	Time (*min*)
apronSep	27	44	0	3.7
oadlSep-float	24	48	0	3.5
oadl-refiner-float	24	48	0	3.7
oadl-seq-W-R-250-float	24	48	0	3.6
eva-basic	30	42	0	3.4
eva-refiner	30	42	0	3.6
predicate	23	49	0	3.7

Table 5.5: Overall Performance of the Analyses on the SV-COMP Benchmarks

do not have any advantages compared the basic **oadlSep-float** configuration. This can be seen in Table 5.5. As these benchmarks rely on non-linear expressions, the performance of the analyses done with configurations of the OADL CPA is quite bad. Out of 74 files only 24 can be analyzed correctly with either of the OADL CPA configurations. The explicit-value analysis is able to analyze 30 programs of this suite correctly, whereas the predicate analysis can only analyze 23 programs successfully. The overall time consumptions of each analysis are quite similar. This is another evidence for the small amount of programs and it also shows that the programs are comparably simple, otherwise the analysis results, as well as the time consumption, would be more divergent. The results of the **apronSep** configuration are once again not taken into consideration for the comparison, due to the bugs in the implementation.

5.4.3 Custom Programs for Showing the Abilities of the OADL CPA

In this section the abilities, such as the handling of relations, or the handling of floating-point variables, of the OADL CPA are illustrated on certain examples. We used CBMC[9] for the comparison. All example programs are safe.

Variable Relations. The program in Listing 5.1 exhibits a linear constraint ($y = x$), such that the assumption $x < -70 || x > 70$ is never satisfiable. The explicit-value

[9] CBMC is a bounded model-checker, c.f. `http://www.cprover.org/`

```c
int main(void) {
  float x = __VERIFIER_nondet_float();
  float y = x;

  if (y < 0) {
    y = -y;
  }
  if (y <= 69) {
    if (x < -70 || x > 70) {
      __VERIFIER_error();
    }
  }
  return 0;
}
```

Listing 5.1: Example program 2

analysis is not able to infer this information, hence it cannot prove this program correct. This is somewhat similar to the problem the explicit-value analysis has by analyzing the program from Listing 2.1. CBMC, the predicate analysis and the OADL CPA are able to prove the program correct.

Floating-Point Variables. The octagon abstract domain is only able to handle linear assignments. Thus every expression which is at least polynomial has to be transformed into an other expression, for example by resolving the value of some variables. The example in Listing 5.2 features many polynomial expressions, the bounds check at the end was chosen such that the *__VERIFIER_error()* call cannot be reached. However, the predicate analysis, as well as the explicit-value analysis report the error location as reachable. The OADL CPA is with its **oadlSep** configuration able to prove the program correct as well as CBMC.

The program in Listing 5.3 does not feature polynomial expressions. It consists simply of a loop which has a high amount of iterations. By using the **oadlWidening** configuration, this program can be analyzed in less than half a minute. The predicate analysis is also able to prove this program correct. Whereas the explicit-value analysis unrolls the loop and therefore does not finish within 900 seconds of run time, CBMC has the same problem.

5.4.4 Conclusion of the Performance Evaluation

When using the merge^{sep} operator the OADL CPA is in many ways equal to the explicit-value analysis. Its basic configuration **oadlSep** is, due to the high memory consumption, not as good as the basic configuration of the explicit-value analysis

```c
float f(float x) {
  return x*x*x;
}

float fp(float x) {
  return x*x;
}

int main()
{
  float IN;
  __VERIFIER_assume(IN > 0.1f && IN < 0.2f);

  float x = IN - f(IN)/fp(IN);

  if(!(x > -0.8 && x < 0.2))
    __VERIFIER_error();

  return 0;
}
```

Listing 5.2: Example program 3

```c
int main(void) {
float step = 0.00000000001;
float value = 0;
int counter = 0;

while (value < 10) {
  counter ++;
  value += step;
}

if (counter < 1000000000000) {
  __VERIFIER_error();
}

return 0;
}
```

Listing 5.3: Example program 4

eva-basic. However when using CEGAR the increase in performance of the OADL CPA is much higher than the one of the explicit-value analysis. Due to not tracking unimportant variables the average memory consumption per verification run can be reduced by approximately 50%. For getting another performance boost, the **oadlWidening-cex** configuration and the **oadl-refiner** configuration can be combined sequentially. In this configuration about 70 programs can additionally be analyzed successfully. The possibility to have that many different configurations with one abstract domain is another advantage of the OADL CPA.

5.5 Restrictions and Challenges

During the implementation and the evaluation of the OADL CPA and the APRON CPA several difficulties were encountered. These are described in the following paragraphs.

Floating-Point Benchmarks. Due to the lack of non-artificial floating-point benchmarks we cannot make any statement about the performance of the OADL CPA and the APRON CPA when proving floating-point programs. The programs that were used for the evaluation suffice to show that both CPAs are able to handle float variables. However their performance on industrial-size samples could not be tested.

The APRON CPA. While the OADL CPA works well, as discussed in the former sections, the APRON CPA still struggles with bugs that render it unusable. Although the API defines some exceptions which are thrown by the library when any problem appears, the behavior of the library regarding certain inputs does not match the intuitive behavior. For example, the library is not directly able to assign a float value to an integer variable. The float value has to be explicitly casted (by adding a cast expression) to an integer. If this cast expression is missing, the library does not throw an exception as it would be expected if a situation occurs that is not supported. Instead, a bottom state is returned and the analysis stops at this path. Finding this bug was quite easy as bottom states may never appear after assignments. However the same applies to assumptions, and in this case it is quite normal that the control flow ends. So the bottom states after assumptions can't be checked. Moreover, while analyzing the integer-related benchmarks from the SV-COMP cases with missing states or program paths appeared, such that false proves occur. As the APRON CPA should similarly to the OADL CPA strictly over-approximate everything that is unknown, false proves should never occur. Hence they are a definite sign for a bug. The short implementation time for the APRON CPA did not suffice to find and fix all bugs. For that reason the APRON CPA is not usable for industrial-sized programs.

6 Conclusion

In this chapter a summary of the thesis is given. Additionally some enhancements that could further improve the performance of the OADL CPA and the APRON CPA will be mentioned.

6.1 Summary

By defining the Octagon CPA the base for verification with octagons in CPACHECKER was provided. The Octagon CPA was formalized as a CPA+ with CEGAR as an additional feature. Besides that sequential combinations of differently configured Octagon CPAs were used to reduce the individual downsides of the single configurations. Two implementations of the Octagon CPA, namely the OADL CPA and the APRON CPA were introduced. They are available in the trunk of the SVN repository of CPACHECKER[1]. Furthermore, the advantages and weaknesses of the implemented CPAs were discussed. Altogether, the OADL CPA is more precise than the explicit-value analysis, because it is able to handle relations between variables. This is the reason for the higher amount of correctly proved programs. Hence, this analysis could also be considered as being a valuable candidate in combination with the predicate analysis or other CPAs.

6.2 Future Work

Starting at this thesis, the OADL CPA and the APRON CPA can be extended in several ways. A first approach would be to reduce the amount of variables saved in each octagon. This can be done by only adding global variables and the variables of the current function to it. At the moment, for successive function calls all variables from the caller function are also part of the octagons in the called functions. This

[1] The SVN repository is available at:
`https://svn.sosy-lab.org/software/cpachecker/trunk`

leads to a higher memory consumption and in some cases it is less precise. For example, when a widening is done due to any loop iteration, the variables from outside of the function are widened equally to those inside the functions, although their values cannot be affected by this loop iteration. The implementation could be done via Block Abstraction Memoization[Won12], which is already a part of CPACHECKER and could be reused. A second approach would be to improve the strategy of the $\mathsf{merge}^{widening}$ operator by using widening with thresholds[2] instead of a normal widening. The introduction of these and further techniques could have great effects on the overall evaluation time and memory consumption. Hence making the octagon-based CPAs more attractive for a combined verification with other CPAs.

[2] A widening with thresholds has the advantage that also loops with a descending variable as condition can be widened to the exact limit. With a normal widening the lower bounds of variables with descending values are always set to $-\infty$. More information can be found in the documentation of the OADL:
`http://www.di.ens.fr/~mine/oct/current/doc/doc_oct.pdf`

Bibliography

[Bey14] Dirk Beyer. *Status Report on Software Verification (Competition Summary SV-COMP 2014)*. Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and of Analysis Systems (TACAS 2014, Grenoble, France, April 5-13), LNCS 8413. Springer-Verlag, 2014.

[BHJM05] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. *The Software Model Checker* BLAST*: Applications to Software Engineering*. International Journal on Software Tools for Technology Transfer (STTT), volume 9, number 5-6, pages 505–525. Springer-Verlag, 2007. Invited to special issue of selected papers from FASE 2004/05.

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

[BHT08] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. *Program Analysis with Dynamic Precision Adjustment*. Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008, L'Aquila, September 15-19), pages 29–38. IEEE Computer Society Press, Los Alamitos (CA), 2008.

[BK11] Dirk Beyer and M. Erkan Keremoglu. *CPAchecker: A Tool for Configurable Software Verification.* CAV, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer-Verlag, 2011.

[BL13] Dirk Beyer and Stefan Löwe. *Explicit-State Software Model Checking Based on CEGAR and Interpolation.* Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013,

Rome, Italy, March 20-22), LNCS 7793, pages 146–162. Springer-Verlag, Heidelberg, 2013.

[BW13] Dirk Beyer and Philipp Wendler. *Reuse of Verification Results: Conditional Model Checking, Precision Reuse, and Verification Witnesses*. Proceedings of the 2013 International Symposium on Model Checking of Software (SPIN 2013, Stony Brook, NY, USA, July 8-9), LNCS 7976, pages 1–17. Springer-Verlag, 2013.

[CH78] Patrick Cousot and Nicolas Halbwachs. *Automatic Discovery of Linear Restraints Among Variables of a Program*. Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

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

[HMM12] Julien Henry, David Monniaux, and Matthieu Moy. *PAGAI: a path sensitive static analyzer*. CoRR, volume abs/1207.3937, . Else4, 2012.

[Min01] A. Miné. *The Octagon Abstract Domain*. AST 2001 in WCRE 2001, IEEE, pages 310–319. IEEE CS Press, October 2001.

[Min06] Antoine Miné. *The octagon abstract domain.* Higher-Order and Symbolic Computation, volume 19, number 1, pages 31–100. 2006.

[Won12] Daniel Wonisch. *Block Abstraction Memoization for CPAchecker - (Competition Contribution)*. TACAS, pages 531–533, 2012.

Eidesstattliche Erklärung

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

Passau, den 10. Juli 2014

Thomas Stieglmaier