MASTER'S THESIS
IN
COMPUTER SCIENCE

AUGMENTING PREDICATE ANALYSIS WITH AUXILIARY INVARIANTS

THOMAS STIEGLMAIER

SUPERVISORS:

PROF. DR. DIRK BEYER
CHAIR OF SOFTWARE SYSTEMS

MATTHIAS DANGL

OCTOBER 9, 2016

ABSTRACT

Predicate analysis is a common approach to software model checking. Abstractions of programs are computed out of predicates found with craig interpolation. The found interpolants are, however, in some cases not ideal, and lead to long-running verification runs. To reduce the reliance on interpolation this thesis evaluates the effects of using separately computed, auxiliary, invariants instead.

Our work is based on the CPA concept, CPACHECKER and the Predicate CPA. It is split into two major parts, on the one hand we introduce a new algorithm for concurrent execution of several analysis in CPACHECKER, as well as communication between such analysis, and on the other hand we show how the Predicate CPA can be augmented with additional formulas in several ways. We chose to evaluate: appending invariants to the precision of the analysis and conjoining invariants either to the path formula or to the abstraction formula. The invariants we want to use are generated by some new approaches directly in CPACHECKER. They can be separated in two classes, on the one hand, the on-the-fly and lightweight invariant generation heuristics which try to find invariants for a certain given program location only, and on the other hand complete analyses, which results are then used for generating invariants for the whole program.

While the lightweight on-the-fly approaches did not yield the expected results, analyses using concurrently computed invariants perform strictly better than comparable analyses without invariants.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACRONYMS

ABE	Adjustable-Block Encoding
BMC	Bounded Model Checking
CEGAR	Counterexample-Guided Abstraction Refinement
CFA	Control-Flow Automaton
CNF	Conjunctive Normal Form
CPA	Configurable Program Analysis with Dynamic Precision Adjustment
LBE	Large-Block Encoding
SBE	Single-Block Encoding
SMT	Satisfiability Modulo Theories

Part I

INTRODUCTION

The following three chapters motivate the usage of invariants and provide the necessary information about other invariant generation and usage approaches.

1

MOTIVATION

Predicate analysis is one of the main approaches in software verification. Its success is based on the recent improvements made to SMT solvers which are mainly used as back-end for solving the formulas created during the analysis.

Another huge enhancement, adjustable-block encoding [BKW10], was invented in 2010 and is implemented in the CPACHECKER framework as a part of the Predicate CPA. With this work we try to further enhance the Predicate CPA by generating and using auxiliary invariants. The invariants should make the analysis faster and less dependent on the interpolation abilities of the SMT solvers. Interpolation is nice to have and easy to use on the one hand, but inappropriate interpolants may lead to loops being unnecessarily unrolled and therefore to longer run times. We try to circumvent this issue by heuristics and separate analyses which generate invariants that can be used as a replacement for interpolation with SMT solvers.

One of the main contributions of this thesis is the development of an algorithm to concurrently run several analyses on the same verification task. Besides that, we formalize different invariant generation and usage strategies,evaluate their impact on a predicate analysis, and show their individual strengths and weaknesses. The concurrent execution of analyses comes hand in hand with the necessity of communication between these analyses. While there was some work on passing information from an earlier running analysis to a later one [BLN+13] no one has extended this to passing results from one concurrently running analysis to another one. This is, however, necessary for computing invariants with one analysis, which should then be used in another analysis.

STRUCTURE OF THIS WORK

This work is structured into three main parts, the first part contains all background information and related work, including in-depth information about the CPACHECKER framework and all kinds of projects using auxiliary invariants to enhance their analyses. The second part is about our additions; we explain different invariant generation heuristics, as well as how they can be used in the Predicate CPA. For more flexibility, we do also create a new algorithm, which allows the concurrent execution of several analyses in one verification run. The third and last part consists of the evaluation, giving detailed insights into our experiments and the advantages and drawbacks we found. We do also mention the restrictions and challenges we had, and finally conclude this thesis with a summary and an outlook on how the process of using invariants can be further extended and improved.

2

BACKGROUND

In this chapter we introduce all theoretical concepts and tools which are used for invariant generation in CPACHECKER. This includes the framework CPACHECKER itself, as well as important analyses and algorithms that are used for evaluation purposes later on.

2.1 PROGRAM REPRESENTATION

A program is represented by a *control-flow automaton (CFA)* [BHT07]. This automaton consists of a set L of locations, which models the program counter (pc), including the initial location $l_0 \in L$ and a set of control-flow edges $G \subseteq L \times Ops \times L$.[1] Ops is the set of program operations. Let X be the set of all program variables, the *concrete state* c of a program assigns a value to each variable from the set $X \cup \{pc\}$. Let C be the set of all concrete states, and let every edge $g \in G$ define the transition relation $\xrightarrow{g} \subseteq C \times \{g\} \times C$ for transforming a concrete state of a certain program location into a concrete state of another program location. The complete transition relation $\rightarrow = \bigcup_{g \in G} \xrightarrow{g}$ is created by unifying all edges. A subset $r \subseteq C$ of concrete states is called region. Now we can define reachability on a CFA.

[1] *The set G models the executed program operations for control-flow from one location to another.*

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

2.2 CONFIGURABLE PROGRAM ANALYSIS

The two main approaches in automated software verification are model checking and program (data-flow) analysis [BHT07]. In contrast to software model checkers, suffering from state-space explosion for large programs, data-flow analyses are usually path-insensitive.

By combining both approaches, the individual drawbacks (false alarms through over-approximation, imprecision due to merging all states for equal locations) can be reduced. On the one hand, the state

space can be shrinked drastically by joining at least some states, and on the other hand, the accuracy of the analysis can be increased by not joining all states but only those with certain (common) attributes (e. g., loop heads).

The original definition of configurable program analysis [BHT07] specifies four components influencing the effectiveness and efficiency of the analysis, the components are called *abstract domain*, *transfer relation*, *merge operator* and *stop operator*. This definition has been extended with an additional *precision* [2] per abstract state and also provides a *precision adjustment function*. The abbreviation for the configurable program analysis with dynamic precision adjustment (CPA) is usually CPA+ yet we stick to calling it CPA [BHT08] for simplicity [3]. More details on the parts of a CPA can be found in the next section.

[2] *A precision can, e. g., be utilized for telling the analysis which variables should be tracked, or that variables should only be tracked up to a certain degree.*

[3] *We do only use CPA+ in this thesis, so this name change does not lead to conflicts.*

2.2.1 *Formalism of a CPA*

A CPA [BHT08] $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ consists of six pieces: 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. These components will be described in the following paragraphs:

- The *abstract domain* $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of three subcomponents. The first two are a set C of concrete states and a semilattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ with
  - a potentially infinite set E of elements, called abstract states,
  - 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 $\sqcup : E \times E \to E$ (the join operator).

  The third part of the abstract domain is a concretization function $[\![\cdot]\!] : E \to 2^C$ which returns the set of concrete states represented by an abstract state.

  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 defines the possible precisions of the abstract domain D. Elements of Π are used by the analysis to keep track of different precisions for different abstract states.

  Let e be an abstract state and π a precision. We call a pair (e, π) *the abstract state e with precision* π.

- The *transfer relation* $\rightsquigarrow \subseteq E \times G \times E \times \Pi$ assigns to every abstract state $e \in E$ with precision π all possible new abstract states e' with precision π for a CFA edge $g \in G$. If $(e, g, e', \pi) \in \rightsquigarrow$ then we write $e \overset{g}{\rightsquigarrow} (e', \pi)$ and furthermore we write $e \rightsquigarrow (e', \pi)$ if an edge g exists with $e \overset{g}{\rightsquigarrow} (e', \pi)$.

  The transfer relation is required to over-approximate all operations for every fixed precision in order to be sound:

$$\forall e \in E, g \in G, \pi \in \Pi : \bigcup_{e \rightsquigarrow (e', \pi)} [\![e']\!] \supseteq \bigcup_{c \in [\![e]\!]} \left\{ c' | c \xrightarrow{g} c' \right\}$$

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

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

  Depending on the abstract state e and the precision π the result of the merge operation can be anything between e' and $\top$ (the result may only be equal to or more abstract than the second parameter). Furthermore the merge operator is not commutative. While the merge operator is not equal to 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 $R \subseteq E$, given as second parameter, is covering the abstract state (given as first parameter) with precision (given as third parameter). To ensure soundness, the termination check has to guarantee that if an abstract state e is covered by the set of abstract states R, every concrete state represented by e corresponds to an abstract state from R:

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

Equivalently 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. An intuitive implementation of the stop operator is

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

(If one abstract state in R is equal to or more abstract than e ($\sqsubseteq$), we say that e is covered by R).

- The *precision adjustment function* $\mathsf{prec} : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$ creates, for a given abstract state e with precision π and a given set of abstract states with precisions, a new abstract state $\hat{e}$ with precision $\hat{\pi}$. 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 soundness requirement for precision adjustment is that the set of concrete states represented by e is a subset of the set of concrete states represented by $\hat{e}$.

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

2.2.2 *The Reachability Algorithm*

In the last section, all necessary components of a CPA were introduced. These components are used by the reachability algorithm [BHT08], which computes, for example, an over-approximation of the set of reachable concrete states for a given initial abstract state with precision and a given CPA. From now on, we will call the reachability algorithm for a CPA `CPAAlgorithm`.

While running the `CPAAlgorithm`, 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 already been found but were not yet processed (frontier), are stored.

The `CPAAlgorithm` computes a set of reachable abstract states with accompanying precision from an initial abstract state with precision. After computing the (intermediate) abstract successors with the transfer relation, these successors are given to the precision adjustment function. The outcome of the precision adjustment function is then merged with every abstract state with precision from the set reached using the given merge operator. If the resulting states are more abstract than those states from the set reached they were merged with, the states from reached are replaced with the new states. 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.

To adapt the `CPAAlgorithm` for usage with CEGAR we have to change the input parameters from *one* initial abstract state with precision to a set R_0 of abstract states with precision. Additionally, a subset $W_0 \subseteq R_0$ of frontier abstract states with precision is given as parameter [BL13]. Algorithm 1 is the resulting reachability algorithm. The function `isTargetState` checks if a state is a target state [4].

[4] *A target state is a state where the specification does not hold. The specification can be implicitly given through the implementation of the CPA.*

Input: a configurable program analysis with dynamic precision adjustment $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$,
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: the set reached and the set waitlist
Variables: a set reached of elements of $E \times \Pi$,
a set waitlist of elements of $E \times \Pi$

```
1  waitlist := W_0
2  reached := R_0
3  while waitlist ≠ ∅ do
4      choose (e, π) from waitlist
5      waitlist := waitlist \ {(e, π)}
6      for each e' with e ⇝ (e', π) do
7          precision adjustment (ê, π̂) := prec(e', π, reached)
8          if isTargetState(ê) then
9              return (reached ∪ {(ê, π̂)}, waitlist ∪ {(ê, π̂)})
10         for each (e'', π'') ∈ reached do
11             combine with existing abstract state
12             e_new := merge(ê, e'', π̂)
13             if e_new ≠ e'' then
14                 waitlist := (waitlist ∪ {(e_new, π̂)}) \ {(e'', π'')}
15                 reached := (reached ∪ {(e_new, π̂)}) \ {(e'', π'')}
16         if ¬ stop(ê, {e | (e, ·) ∈ reached}, π̂) then
17             add new abstract state
18             waitlist := waitlist ∪ {(ê, π̂)}
19             reached := reached ∪ {(ê, π̂)}
20 return (reached, ∅)
```

Algorithm 1: `CPAAlgorithm` [BHT08, BL13]

2.2.3 *Composite Program Analysis*

With the concept of a CPA we can now define analyses for tracking explicit values of variables throughout the control-flow, or we can track the values of variables as intervals or even boolean formulas. All of these analyses need to take care of the different locations of the CFA and most probably also of the call-stack. For separation of concerns we do now introduce the possibility to combine several CPAs into one composite CPA:

$$\mathcal{C} = (\mathbb{D}_1, ..., \mathbb{D}_n, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times) \text{ with } n \in \mathbb{N}$$

This way, we can split up the responsibilities of single CPAs and make their purpose clearer [5] [BHT08]. Such a composite CPA consists of a finite amount of CPAs and:

[5] *We can, e. g., make one CPA which is solely responsible for tracking the location, and one for tracking the call stack.*

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

Let $i \in [1; n]$, the five composites above are expressions over the components of the involved CPAs $(\Pi_i, \rightsquigarrow_i, \mathsf{merge}_i, \mathsf{stop}_i, \mathsf{prec}_i, [\![\cdot]\!]_i, E_i, \top_i, \bot_i, \sqsubseteq_i, \sqcup_i)$. There are also two new operators:

$$\text{strengthen: } \downarrow\colon \times_{i=1}^{n} E_i \to E_1$$
$$\text{comparison: } \preceq \subseteq \times_{i=1}^{n} E_i$$

Strengthening is an additional operator for a CPA which can be used as part of a composite. Its purpose is to compute a stronger element from the lattice set E_1 by using the information from an element of the lattice sets $E_2 ... E_n$; $\downarrow (e_1, ..., e_n) \sqsubseteq e_1$ has to be fulfilled. The comparison operator allows us to compare elements of different lattices.

For a composite analysis $\mathcal{C} = (\mathbb{D}_1, ..., \mathbb{D}_n, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$ 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 $\Pi_\times = \Pi_1 \times ... \times \Pi_n$ and the components of the product domain $D_\times = D_1 \times ... \times D_n = (C, \mathcal{E}_\times, [\![\cdot]\!]_\times)$ are defined by the product lattice $\mathcal{E}_\times = \mathcal{E}_1 \times ... \times \mathcal{E}_n = (E_1 \times ... \times E_n, (\top_1, ..., \top_n), (\bot_1, ..., \bot_n), \sqsubseteq_\times, \sqcup_\times)$ with $(e_1, ..., e_n) \sqsubseteq_\times (e'_1, ..., e'_n)$ iff $\forall i \in n : e_i \sqsubseteq_i e'_i$ and $(e_1, ..., e_2) \sqcup_\times (e'_1, ..., e'_n) = (e_1 \sqcup_1 e'_1, ..., e_n \sqcup_n e'_n)$ and the product concretization function $[\![\cdot]\!]_\times$ in such a way, that $[\![(d_1, ..., d_n)]\!]_\times = [\![d_1]\!]_1 \cap ... \cap [\![d_n]\!]_n$ is met.

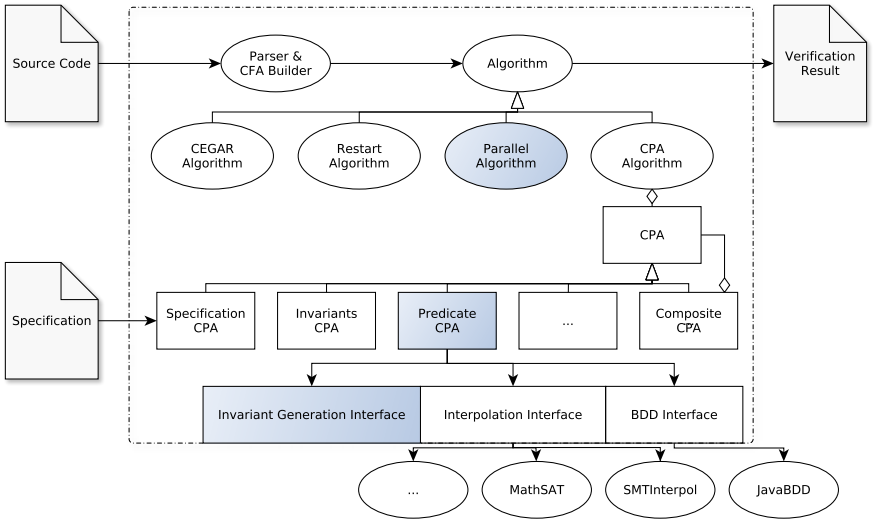

Figure 1: CPACHECKER architecture [BK11]

2.3 CPACHECKER

CPACHECKER [6] is a software verification framework based on the concepts of CPA [BK11]. It is published under the Apache 2.0 license. The program analysis is performed by the implemented CPAs. These CPAs can be combined freely, either for usage in a composite analysis (cf. Section 2.2.3) or for a sequential usage (cf. Section 2.3.3). C and Java are the programming languages which CPACHECKER is able to analyze. But while for Java the support is quite basic, the main focus lies on the evaluation of C programs.

[6] *More information and the sources can be found at* `cpachecker.sosy-lab.org/`

2.3.1 *Basic Architecture*

In Figure 1 the basic CPACHECKER architecture is shown. The highlighted parts are especially important for this thesis, for example the Parallel Algorithm is even added in this thesis, but already in the figure to show where it is located compared to other CPACHECKER components.

A simple verification run could work as follows: at first the source code is parsed [7], then a CFA is created and afterwards the result is computed by the `CPAAlgorithm` with the configured CPAs. The result is then given to the user of CPACHECKER.

[7] *We use the Eclipse CDT for that purpose, it can be found at* `eclipse.org/cdt/`

2.3.2 *Composite CPAs in* CPACHECKER

The concept of a composite analysis, introduced in Section 2.2.3, enables us to separate the different concerns of component analyses from each other. The component CPAs can then be combined on demand. Two important features for every analysis are tracking the call-stack and the program counter. So, instead of repeatedly implementing tracking of the location and the call-stack for every analysis, one can now create separate CPAs for modeling the call-stack and tracking the program counter.

2.3.3 *Sequential Combination of Analyses*

Sequentially combining several separate analyses is a concrete implementation of conditional model checking [BHKW12] (cf. Section 3.5). This approach is also implemented in CPACHECKER in the `RestartAlgorithm`. Whenever the result of a verification run is not *safe* or *unsafe*, the next configuration is started with the input condition FALSE and has to verify the program without any initial assumptions again. Furthermore it is possible to skip subsequential analyses based on the outcome of earlier ones, for example if an analysis finds out that the program contains concurrency, all analyses which do not support concurrency can be omitted, prohibiting the model checker from consuming time with analyses that do not provide the necessary abilities.

2.3.4 *Counterexample-Guided Abstraction Refinement*

Counterexample-Guided Abstraction Refinement (CEGAR) [CGJ+03] is a technique that tries to overcome the state space explosion in model checking by abstracting unnecessary information. This is done by iteratively refining the precision of the analysis each time an infeasible counterexample is identified. The necessary information for refining the precision can be extracted by several techniques out of the infeasible counterexample. Some possibilities are for example Craig interpolation [BK11], path invariants (cf. Section 2.4) or heuristics that extract the precision increment from program statements, e. g., assumptions.

While the original approach is only aimed at symbolic model checking, CEGAR has been extended to also work with explicit-state model checkers [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 $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$

```
1  reached := {(e_0, π_0)}
2  waitlist := {(e_0, π_0)}
3  while TRUE do
4      (reached, waitlist) := CPAAlgorithm(𝔻, reached, waitlist)
5      if waitlist = ∅ then
6          return safe
7      else
8          σ := extractErrorPath(reached)
9          feasible error: report bug, else refine and restart
10         if isFeasible(σ) then
11             return unsafe
12         else
13             π := π ⋃ refine(σ)
14             reached := (e_0, π)
15             waitlist := (e_0, π)
```

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

In Algorithm 2 a simple CEGAR algorithm working in combination with a CPA is displayed. The method `extractErrorPath` extracts the found counterexample out of the set reached. The feasibility of the counterexample is tested with the method `isFeasible`. If the counterexample is feasible we can stop the analysis and return the found property violation to the user. When the counterexample is infeasible we use the procedure `refine` to refine the precision of the analysis.

The CEGAR algorithm is implemented in CPACHECKER, and further more has an additional option which delays the refinement until the state space is fully explored with the current precision. When this happens the refinement starts and all error locations are handled at once. The latter approach will be used later on for invariant generation.

2.3.5 *The Predicate CPA*

The Predicate CPA [BKW10] is based on (boolean) predicate abstraction[8]. Let $\mathcal{P}$ be a set of predicates over program variables, a formula φ is a boolean combination of predicates from $\mathcal{P}$. We call π a precision for formulas with $\pi \subset \mathcal{P}$. Π is a precision for programs given by the function $\Pi : L \to 2^{\mathcal{P}}$, which assigns a precision for formulas to each program location. The strongest boolean combination of predicates from precision π entailed by φ is called boolean predicate abstraction $(\varphi)^{\pi}$ of a formula φ. The outcome of a predicate abstraction can be used as abstract state and represents a region of concrete program states. The computation of the predicate abstraction can be done by satisfiability modulo theories (SMT) solvers. Therefore we introduce a propositional variable v_i for each predicate $p_i \in \pi$ and then ask the SMT solver for satisfying assignments for the formula $\varphi \wedge \bigwedge_{p_i \in \pi}(p_i \iff v_i)$. The disjunction of all conjuncted satisfying assignments is the result of the boolean predicate abstraction. Computing the successor φ' of φ is done by applying the abstract strongest post operator for predicate abstraction with a program operation op. The strongest post operator can be defined as $\varphi' = (\mathtt{SP}_{op}(\varphi))^{\pi}$, where SP denotes the strongest post condition operator, which is applied first, afterwards, the result is used for computing the boolean predicate abstraction.

[8] *While other abstraction methods, such as cartesian abstraction, are also possible, the default abstraction method is boolean predicate abstraction.*

The original predicate abstraction [BKW10] works either with single-block encoding (SBE) or with large-block encoding (LBE) and while SBE leads to a slower analysis, we need to preprocess the analyzed program for LBE. Both approaches are unified in adjustable-block encoding (ABE), an approach which choses dynamically whether an abstraction should be computed [BKW10]. For adding this as a feature to the traditional predicate abstraction, we store two separate formulas in each state, an abstraction formula ψ and a path formula φ. States at which an abstraction is done are called *abstraction states*, all other states are called *non-abstraction states*. Both are disjunct types of abstract states of this CPA. Program paths between two abstraction computations may consist of many CFA edges where states for locations inside such paths are always non-abstraction states. For these non-abstraction states the strongest post condition is stored in the path formula of each state, while the abstraction formula remains unchanged. At abstraction states, a new abstraction formula is computed. The decision when to do abstraction is done by the block-adjustment operator `blk` which returns FALSE if no abstraction should be computed for a given pair of an abstract state e and a CFA location l, and TRUE otherwise.

By adjusting the block size on demand, we can have many concrete configurations lying in between SBE and LBE and even block sizes larger than those produced with LBE are possible. The Predicate CPA with ABE is defined as follows [9]:

[9] *Please note that the location is not modeled within this CPA but is still needed, so having a composite with a CPA for location tracking is necessary.*

- The abstract domain $D_{\mathbb{P}} = (C, \mathcal{E}, [\![\cdot]\!])$ is given by the semi-lattice $\mathcal{E} = (2^{\pi}, \text{TRUE}, \text{FALSE}, \sqsubseteq, \sqcup)$, where the partial order $\sqsubseteq \subseteq E \times E$ is defined as $e_1 \sqsubseteq e_2 \iff (e_2 = T) \vee (\psi_1 \wedge \varphi_1 \Rightarrow \psi_2 \wedge \varphi_2)$ and the join operator $\sqcup : E \times E \to E$ is defined as the least upper bound of both operands, according to the partial order. The concretization function is given by $[\![e]\!] = \{c \in C \mid c \models \varphi_e\}$.

- The set Π of precisions contains the predicates used for predicate abstraction. It is initially empty, and combined with CEGAR upon finding infeasible errors, we compute the necessary precision increment to refute the infeasible counterexample using Craig interpolation [Cra57].

- The transfer relation $\rightsquigarrow \subseteq E \times G \times E \times \Pi$ computes the abstract successor $e' = (\psi', \varphi')$ for an abstract state $e = (\psi, \varphi)$ and a CFA edge $g = (l, op, l')$ such that $\varphi' = \mathtt{SP}_{op}(\varphi) \wedge (\psi' = \psi)$ holds.

- The merge operator $\mathsf{merge} : E \times E \times \Pi \to E$ is defined as follows for two states $e_1 = (\psi_1, \varphi_1)$ and $e_2 = (\psi_2, \varphi_2)$:

$$\begin{cases} e_2 & \text{if this is an abstraction location} \\ e_2 & \text{if } \psi_1 \neq \psi_2 \\ (\psi_1, \varphi_1 \vee \varphi_2) & \text{otherwise} \end{cases}$$

- The stop operator is stop^{sep}.

- The precision adjustment function $\mathsf{prec} : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$ creates for a given abstract state e with precision π and a given set of abstract states with precisions a new abstract state $\hat{e}$ with precision $\hat{\pi}$ depending on `blk`. The program location l, which is necessary for `blk`, can be retrieved from another CPA that tracks the location and is part of the composite analysis. While the computed abstract state $\hat{e}$ may be different to e, the precision stays the same:

$$\begin{cases} \hat{e} = ((\mathtt{SP}_{op}(\varphi \wedge \psi))^{\pi}, \text{TRUE}) & \text{if } \mathtt{blk}(e, l) \\ \hat{e} = e & \text{otherwise} \end{cases}$$

2.3.6 *The Invariants CPA*

In contrast to the Predicate CPA, the Invariants CPA [BDW15b] does not use SMT solvers, but is based on expressions over intervals. The important parts of this CPA will be introduced in the following paragraph:

- The abstract domain of the Invariants CPA is based on expressions over intervals. Abstract states in this domain are mappings $M : X \to Expr$ from a set of program variables X to a set of arithmetic expressions $Expr$, where $Expr$ can consist of unary and binary expressions $U = \{\neg, \sim, -\}$ and $B = \{+, *, /, \%, =, <, \hat{}, |, \vee, \&, \wedge, \gg, \ll, \cup\}$, as well as program variables or disjunctions of intervals I of the form $[u, l]$ with $u, l \in \mathbb{Z} \cup \infty$. The (recursive) definition is $Expr \subseteq ((Expr \times B \times Expr) \cup (U \times Expr) \cup X \cup I)$.
- The set of precisions Π contains precisions $\pi = (Y, n, w)$ with $Y \subseteq X$, a maximal expression nesting depth $n \in \mathbb{N}$ and a boolean flag $w \in \mathbb{B}$ specifying whether widening should be used. All abstract states have the same precision. In general, the Invariants CPA is tracking all program variables, but most of them are over-approximated while joining states. Y is a selection of important program variables, which are not over-approximated while joining states. n specifies the accuracy of inter-variable relations. With w set to TRUE widening is used to sacrifice accuracy for efficiency. This is especially important for programs with many loop iterations.
- The merge operator $\mathsf{merge} : E \times E \times \Pi \to E$ is defined as following for two states e_1 and e_2:

$$\begin{cases} \texttt{widen}(e_1, e_2) & \text{if } w \wedge \neg \texttt{differ}_\pi(e_1, e_2) \\ \texttt{union}(e_1, e_2) & \text{if } \neg w \wedge \neg \texttt{differ}_\pi(e_1, e_2) \\ e_2 & \text{otherwise} \end{cases}$$

  `differ` is a function that checks if the expressions over the important variables Y are equal in both states, if not, we do not merge at all. A widening is done according to w, where widening means that for each variable only a single (potentially infinite) interval is assigned. `union` is the union of all values for each variable.

While the precision is fixed for a complete verification run it can be configured to be continuously-refined by using Algorithm 3 as a wrapper around Algorithm 1. With this wrapper algorithm, only safe

programs can be found, for all other programs, the result will be unknown [10]. For example, the first iteration of doing an analysis with the Invariants CPA is done with an empty set of important variables Y, and an expression nesting depth n of 1. With each iteration we can now increase n as well as inserting variables into Y. If at some time, no state violating the specification is in the reached set (indicated by the method `containsTargetState`) Algorithm 3 terminates and tells the user that the program is safe.

[10] *Due to the fixed precision we do not know if a bug was found because of being to coarse or because the bug actually exists.*

An additional feature of Algorithm 3 is that one can extract invariants from it. This is for example necessary for k-induction-based analyses (cf. Section 2.5). `getCurrentlyKnownInvariants` is the name of the corresponding function.

Input: a configurable program analysis with dynamic precision adjustment $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$,
a set of initial abstract states E,
an initial precision π_0
Output: TRUE if no target state is found
Variables: a set reached of elements of $E \times \Pi$,
a precision π,
an invariant Inv

1 $\pi := \pi_0$
2 $Inv :=$ TRUE
3 **Loop**
4 reached := `CPAAlgorithm`$(\mathbb{D}, \{(e, \pi) | e \in E\}, \{(e, \pi) | e \in E\})$
5 **if** $\neg$`containsTargetState`(reached) **then**
6 **return** TRUE
7 $Inv := Inv \wedge \bigvee_{s \in reached} s$
8 $\pi :=$ `RefinePrec`(π)

Algorithm 3: Continuous Precision Refinement and Invariant Generation [BDW15b]

2.4 PATH INVARIANTS

A path invariant [BHMR07] is an invariant created for a path program — the smallest syntactic subprogram containing an infeasible error path. A path program may contain loops and therefore often represents a group of infeasible error paths that would be found upon unrolling the loop. By computing invariants capable of refuting more than one infeasible error path, a weakness of CEGAR, loops leading to a potentially infinite amount of necessary refinements, [11] can be overcome.

[11] *This happens, e. g., by choosing disadvantageous precision increments, such that loops have to be unrolled and the infeasible error is found again in each loop iteration.*

By combining CEGAR with invariant generation and using the generated invariants, for example, as precision increment instead of interpolants, we are able to reduce the number of necessary refinements, and therefore lower the analysis time. This approach was initially implemented in CPAchecker as a term paper.[12] Although the approach worked, there were some conceptual issues, which will be addressed in this masters thesis. The implementation of path invariants in CPAchecker for the Predicate CPA was done using Algorithm 3 without having multiple iterations, but stopping after the first one.[13] The computed invariants are retrieved via `getCurrentlyKnownInvariants` and appended to the precision of the analysis instead of computing interpolants. Due to the restriction of the invariant generation to a certain path of the program, the generated invariants do not hold for the complete program, but only for the given path program, which prevents, for example, directly conjoining path invariants to the abstraction formula in a Predicate CPA. Instead, we can only add them to the precision of the analysis.

[12] *See `stieglmaier.me/uploads/invariants.pdf` for more details.*

[13] *Path invariants are computed when they are needed, so continuously-refining the precision of the analysis takes too much time as it is not running in parallel and has potentially no end. By using Algorithm 3 we can access the method for retrieving invariants which is the reason for using it.*

2.5 *k*-INDUCTION WITH CONTINUOUSLY-REFINED INVARIANTS

k-induction is a model-checking approach which extends traditional bounded model checking (BMC) based strategies, such that they are not only able to find bugs, but also to prove safety. BMC is used in *k*-induction to unroll the program until a certain limit *k* for the length of the path is reached. If an error is found the analysis is finished. If not we try to verify the program by induction. When this fails, we increase *k* and start over with BMC. CPAchecker uses split-case *k*-induction[14] and therefore we focus on it for this work [BDW15a]. The following sections provide more details about the theory of *k*-induction and how it can be implemented in a model checker.

[14] *Another approach is combined-case k-induction, where the base and the step case of the induction are not separated.*

2.5.1 *Bounded Model Checking*

BMC is a technique for software falsification. By setting a limit *k* to the length of the unrolling of a program, only counterexamples up to a certain length can be found. SAT or SMT solvers can be used to check the satisfiability of unrolled paths through a program. BMC in combination with the Predicate CPA can be done by setting `blk` to do no abstraction until a certain bound is met (instead of doing an abstraction, e. g., for all loop heads). Due to the given bound this approach is not able to make statements about the safety of a program, but instead only found errors can be reported.

2.5.2 *k-Induction*

k-induction uses BMC to check for the presence of counterexamples regarding a certain safety property P. If no counterexamples exists in a path unrolled up to a length k we try to verify the program by induction. Consider a program with a loop: if P holds for $k = 1$ this means that no violation of the property P exists when unrolling exactly one iteration of the loop, however a counterexample in one of the following loop iterations could still exist. The safety property P is given by:

$$P(l, f) = \neg(\exists s \in \mathsf{reached} : \mathtt{loc}(s) = l \wedge f)$$

It depends on a location l and a formula f.[15] The property holds as long as no state s is reachable (i.e. exists in the set reached) such that the location of s ($\mathtt{loc}(s)$) is equal to the error location l.

[15] *When searching for errors in the program, l will be the error location and f is simply* TRUE.

If we are able to prove that for any given iteration through the loop P is not violated, and P also holds in the following iteration, we may be able to prove safety of the analyzed program. If the inductiveness check fails, we can increase k and try again. This is called *iterative-deepening k-induction* [BDW15a]. Algorithm 4 shows the iterative deepening, and the separation of the base and the step case. In the following paragraphs, the algorithm will be explained in more detail.

BASE CASE The base case consists of running BMC with the current bound k. As described in Section 2.5.1, this unrolls all paths through the program from initial program states denoted by the predicate I up to a maximum amount of loop iterations k. If the formula in line 3 of Algorithm 4 is satisfiable there exists a counterexample with a length of at most k.

FORWARD CONDITION If the base-case formula is unsatisfiable we can check whether there exists a path with a length greater than k or whether we have fully explored the state space of the program. This check is called *forward_condition* and can be found in line 6. If the state space is fully explored, the program is safe[16] and the algorithm terminates.

[16] *Besides proving safety of programs we can also check predicates on inductiveness, cf. the paragraph on checking inductiveness of formulas.*

STEP CASE In the step case we check that after any sequence of k loop iterations without a counterexample there is also no counterexample in the loop iteration $k + 1$. This check is necessary if the forward condition is not satisfiable. By leaving out the *step_case* computation we would be using only BMC with continuously increasing k, such

Input: an initial value $k_{init} \geq 1$ and an upper limit k_{max} for the bound k,
a function $\texttt{inc} : \mathbb{N} \to \mathbb{N}$ with $\forall n \in \mathbb{N} : \texttt{inc}(n) > n$ for increasing the bound k,
the initial states defined by the predicate I,
the transfer relation defined by the predicate T,
and a safety property P
Output: TRUE if P holds, FALSE otherwise
Variables: the formulas *base_case*, *forward_condition* and *step_case*, an invariant Inv, a bound k

1 $k := k_{init}$
2 **while** $k \leq k_{max}$ **do**
3 $\quad base\_case := I(s_0) \wedge \bigvee_{n=0}^{k-1} \left(\bigvee_{i=0}^{n-1} T(s_i, s_{i+1}) \wedge \neg P(s_n) \right)$
4 $\quad$ **if** $\texttt{sat}(base\_case)$ **then**
5 $\quad\quad$ **return** FALSE
6 $\quad forward\_condition := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$
7 $\quad$ **if** $\neg\texttt{sat}(forward\_condition)$ **then**
8 $\quad\quad$ **return** TRUE
9 $\quad step\_case_m := \bigwedge_{i=m}^{n+k-1} (P(s_i) \wedge T(s_i, s_{i+1})) \wedge \neg P(s_{n+k})$
10 $\quad$ **repeat**
11 $\quad\quad Inv := \texttt{getCurrentlyKnownInvariants}()$
12 $\quad\quad$ **if** $\neg\texttt{sat}(\exists n \in \mathbb{N} : Inv(s_n) \wedge step\_case_n)$ **then**
13 $\quad\quad\quad$ **return** TRUE
14 $\quad$ **until** $Inv = \texttt{getCurrentlyKnownInvariants}()$
15 $\quad k := \texttt{inc}(k)$
16 **return** unknown

Algorithm 4: Iterative-Deepening k-Induction [BDW15a]

that safety of programs could be proved when the *forward_condition* holds. We want the analysis to not unroll the complete state space, and hope that the inductive step succeeds at some point. This check will however often fail when model checking of software is done, as the state space — for which the property should hold — consists typically not solely of relevant states, but also of unreachable states for which the property does not hold.

For example, if we consider a loop with a loop counter which has only positive values, by using induction we try to prove the property for all values, not only positive ones, and therefore the check fails (the necessary information — the loop counter has only positive values — may not be available in the induction hypothesis, which leads to the failing check). To overcome this problem, we can add auxiliary invari-

ants to the satisfiability check of the step case formula. This can be seen from line 9 to line 14.[17] If the conjunction of the auxiliary invariant and the step-case formula is unsatisfiable we have proved the program to be safe, otherwise we are not able to draw a conclusion about the safety of the program with the current value of k. By increasing k (cf. line 15) and running Algorithm 4 again from line 2, we try to prove the program iteratively again.

[17] *The* repeat-until *loop is rerun as long as more precise invariants can be found during the satisfiability computation of the step case.*

AUXILIARY INVARIANTS Auxiliary invariants are a key feature for using k-induction for software model checking. In the scope of Algorithm 4 they can be generated concurrently, for example with Algorithm 3 and then retrieved when they are needed for the step-case computation. This analysis may be able to prove the safety of the program itself but this is not the main purpose of the invariant generator.

CHECKING THE k-INDUCTIVNESS OF A FORMULA In addition to proving safety of programs we can also check the inductiveness of a given predicate *candidate_invariant* for a program by setting $l = \mathit{invariant\_location}$ and $f = \neg \mathit{candidate\_invariant}$ for $P(l, f)$. With $k = 1$ we check 1-inductiveness of the given predicate, no auxiliary invariants are needed for this.

3

RELATED WORK

A typical area where auxiliary invariants are used is software verification with *k*-induction-based model checkers [BDW15a, AS06, KT11]. Other than that invariants can be combined with CEGAR [BHMR07] or they are computed in a separate (potentially parallel) analysis solely for the purpose of improving the main analysis [GKN15].

The invariant generation itself is a separate process which is integrated into the software verifiers. While there exist some potentially usable invariant generators [GR09, EPG+07, AS06] they are either written for other programming languages like Lustre [HCRP91] or they are not yet mature enough for analyzing real-world C programs (cf. Section 3.4). The only reasonably working invariant generation for our case is provided by CPACHECKER itself, and was initially implemented for continuously-refined invariants used together with *k*-induction [BDW15a].

3.1 MODEL CHECKERS USING INVARIANTS

In practice, PKIND[18] and some configurations of the CPACHECKER framework often need auxiliary invariants to make the analysis terminate at all. This is due to a general problem with *k*-induction-based verifiers: *k*-induction itself does not distinguish between reachable and unreachable parts of the space space of a program [BDW15a][19], but safety properties often do not hold in unreachable parts of the state space.

[18] PKIND *is a model checker based on k-induction.*

[19] *For more information on this problem, see Section 2.5.*

Invariant generation running in parallel to the model checker was introduced by PKIND. Their invariant generation is also based on *k*-induction, which is used to check *candidate invariants* synthesized out of predefined templates. To leverage the advantages of parallelism, *k*-induction for invariant generation is set up to firstly check 0-inductivity and return the valid invariants, and then continuously increase *k* returning the newest invariants found for each *k* [KT11]. A comparable approach was also implemented in CPACHECKER and furthermore another invariant generation strategy based on a data-flow analysis was added [BDW15a].

2LS [BJKS15] is a tool that is based on BMC, *k*-induction and abstract interpretation. The three verification approaches are combined such that with abstract interpretation, invariants are generated out of given templates, and these invariants are used for *k*-induction. If an error location is found to be reachable, it is double-checked with BMC.

SEAHORN [GKN15] is a program-verification framework implemented in LLVM[LA04]. It converts LLVM bitcode to horn clauses and then, uses the PDR / IC3 algorithm with the SMT solver Z3 [HB12] to verify the safety of the program. Additionally the IKOS [BNSV14] library can be used to generate invariants from the LLVM bitcode, which are then also encoded as horn clauses and added to the program that should be verified. According to their evaluation, the additional invariants improve the verification process such that some tasks that ran into timeouts before (without auxiliary invariants), can be successfully verified.

DAFNY [LM10] is a programming language which has built-in support for specifications. These specifications are part of the code and they are used for verifying the correctness of the corresponding program with the DAFNY static program verifier. This verifier is run as part of the compiler, and only if the code was successfully verified, a binary is created. The given specifications can be seen as invariants given by the programmer. This is also the main difference to the aforementioned tools: invariants are not computed automatically, but instead they are given by the user. If the compiler is not able to prove the given invariants, it stops and asks for a more concise specification, for example, the split of one specification into several lemmata can help the compiler check the specification.

In contrast to *k*-induction-based model checkers, where the invariants are strictly needed, this work aims at creating and using invariants with analyses that do not need them, comparably to SEAHORN. They can then be used to replace interpolants up to a certain degree, or to just have some additional formulas to strengthen states at certain locations with the aim of speeding up the analysis. Unlike DAFNY we do not need user-interaction but instead completely rely on automatic invariant generators.

3.2 PATH INVARIANTS

Path invariants are another approach to creating lightweight invariants. The idea is to not use whole programs for invariant generation, which in most cases is very costly, but instead generating invariants only for small subprograms, by combining invariant generation with

CEGAR. If a found error location is known to be infeasible, a path program — a semantically correct program, consisting only of the error path and all (potentially unrolled) loops in it — is created, which is then used for invariant generation [BHMR07]. The generated invariants are only invariants for this specific path program and not for the whole program, and thus they can generally not be used in all cases where real invariants could be used (cf. Section 2.4).

A first approach on implementing path invariants in CPACHECKER exists[20], its capabilities and the usability were greatly enhanced for this master's thesis.

[20] *The implementation of path invariants in* CPACHECKER *was done by me during a seminar on Software Verification, it can be found at* `stieglmaier.me/projects.html`.

3.3 LOOP ACCELERATION

Finding compact but still sufficiently precise loop invariants is a struggle for real-world C programs. In many cases, loops are unrolled which gets more ineffective with increasing loop sizes. A technique for summarizing loops is *acceleration*. At first, a closed-form representation of the loop-behavior is computed, which is then turned into an *accelerator* — a code snippet, skipping intermediate loop states to the loop end in one step. While in general, finding accelerators is as difficult as the verification problem itself, restricting the acceleration to some special cases, for example, linear loops [JSS14], makes it a good addition for program analyzers [MWK+15]. The accelerator is not an invariant itself but supports the invariant synthesis done by program analyzers. Loop acceleration can either be done as a preprocessing which results in a new, instrumented, code file, or during the analysis as it is done with Aspic and C2fsm [FG10].

Loop acceleration is a heuristic that is used to support program analyzers just like we evaluate the usage of lightweight invariants for this case. A combination of both approaches is future work.

3.4 OTHER INVARIANT GENERATORS

Besides the directly mentioned invariant generation approaches in the last two sections, there exist several standalone tools generating invariants for certain programming languages:

INVGEN [GR09] is an automatic linear-arithmetic invariant generator for imperative programs. Invariants are synthesized at each cut-point location (for example at loop entries) out of templates, consisting of parameterized linear inequalities over program variables. INVGEN takes as input a set of transition relations written in Prolog syntax. C

is only supported partially by a frontend which converts a subset of C — neither function calls, nor arrays and pointers are supported — to the required input language.

DAIKON [EPG+07] is a dynamic detector of likely invariants. Before running the program it is instrumented and during the runtime the computed values are observed. This results in invariants that hold for the execution of this single run, by for example changing the user input of the program the found invariants may change. In contrast to INVGEN, this approach fully supports C, the only drawback is the lacking support for non-determinism, another essential part of almost every program (e. g., user input, sensor data) [21].

[21] *This does also mean that the found likely invariants are only applicable for a given user input.*

There exist much more invariant generators using different techniques, e. g., abstract interpretation [LB04] or abduction [DDLM13], another one is based on assertions in the program [Jan07]. All these approaches have in common that they are bound to a specific input language, and the implementation of these approaches do mostly also only support this language without any extensions, making them unusable for invariant generation of real-world C programs.

Invariants can also be generated with CPACHECKER by running an analysis and afterwards analyzing the set of reached states. The disjunction of all states for a location builds the invariant for that location if the analysis was sound and every possible state was explored [BDW15a]. This approach is language agnostic [22], easy to use and fulfills all requirements needed for this work. It will be used and extended within this master's thesis.

[22] *The only limit for languages we have is given by the supported languages of* CPACHECKER*, which are currently C and Java. The technique itself is applicable to any language.*

3.5 CONDITIONAL MODEL CHECKING

While in traditional model checking, the result of a verification run is either *safe* or *unsafe* [23], in conditional model checking [BHKW12], the result is a condition Ψ under which the analyzed program satisfies a given specification. This is helpful in case of failures, as the consumed resources are not wasted, but instead the output conditions may speed up subsequent verification runs. For example, when a timeout occurs, the model checker could summarize the successfully analyzed part of the program in the output condition by declaring that as long as the program execution stays within this part, the program is safe. For a complete analysis, *safe* is represented by $\Psi = \text{TRUE}$ and *unsafe* is represented by $\Psi = \text{FALSE}$. While this approach has not much in common with the original idea of invariants, it is quite close to path invariants (information for a certain path of a program is computed by a one analysis and used by another analysis) and sequentially combined analyses [24] will be used for invariant generation in this thesis.

[23] Unknown *may be a valid result, too, in case of timeouts or other problems.*

[24] *For these analyses Ψ will always be* FALSE*, but some other information computed in earlier analyses is passed to the next ones.*

Part II

GENERATING AND USING AUXILIARY INVARIANTS IN CPACHECKER

The following two chapters give detailed information about conceptual changes and additions that had to be made, as well as a documentation of the most important features that were added to CPAchecker.

4

CONCEPTUAL EXTENSIONS

In this chapter, we first give an overview over the current state of invariant generation and usage in CPACHECKER. Then, we introduce some new concepts to improve the handling of invariants and also explain some additional approaches to invariant generation.

4.1 ARCHITECTURE BEFORE THIS THESIS

Before this master's thesis, auxiliary invariants were mainly used for analyses doing *k*-induction in CPACHECKER. The only other use-case were path invariants, which also rely on Algorithm 3 for generating invariants. In Figure 2, the most important parts for generating and retrieving invariants in CPACHECKER are displayed.

There are several implementations of the `InvariantGenerator` interface. First there is the `CPAInvariantGenerator`, a class that uses a given CPA and Algorithm 1 without the possibility of adding CEGAR or continously-refined invariants.

Then there is the `AdjustableInvariantGenerator`, which can be wrapped around any `CPAInvariantGenerator`, and more importantly, which can be used to adjust some conditions of the invariant generation, for example, resetting the reached set to only contain the initial state, and increasing the precision before restarting the `CPAAlgorithm`. The `AutoAdjustingInvariantGenerator` is a wrapper around an `AdjustableInvariantGenerator`. With this

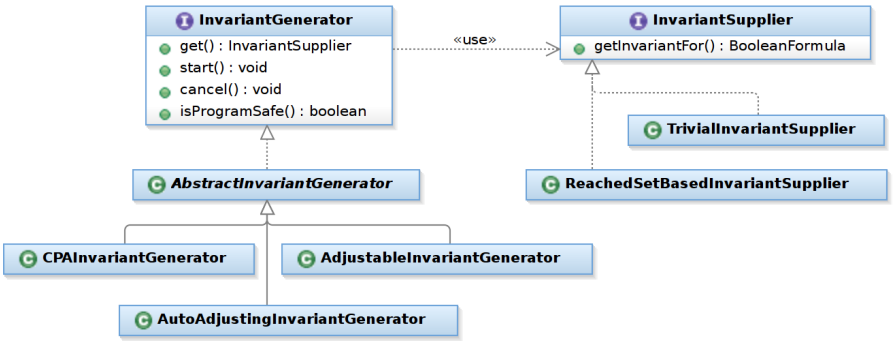

Figure 2: Invariant generation in CPACHECKER (old)

implementation, the given function to adjust the invariant generation is called automatically upon a finished invariant generation run, and then invariant generation is started again. This is done in a loop until either the invariant generation is cancelled or the invariant generator proved the safety of the program.

Besides the `CPAInvariantGenerator`, all invariant generator implementations are package private and therefore hidden from users. Using them is only possible via the `CPAInvariantGenerator` by setting the corresponding configuration options. Moreover, invariant generation can be either executed sequentially, or it can be run in parallel on a separate thread [25]. The method `isProgramSafe` indicates if the invariant generator was able to prove the safety of a program. In the case that safety was proved by the invariant generator, we can stop the overall analysis and return that the program is safe. This is not easily possible, as according to Algorithm 1 the returned value of an analysis is its reached set, which either contains an error state (the program is unsafe) or does not contain an error state (the program is safe). From inside the CPA we can however not change the returned reached set. This is only possible in the algorithm. Therefore, the only possible option is to remove all currently contained error states from the reached set of the CPA and additionally removing all states from the waitlist. A weakness of this approach is, that the returned reached set does not contain the information that the specification violations in the the program are not reachable. And even worse the violations would be found again if we do not manually remove all pending states from the waitlist [26]. Thus, we have an invalid reached set as result of the analysis in the case we want to use the shortcut as soon as the invariant generator proved safety.

[25] *For both options calling the `start` method starts the invariant generation.*

[26] *Instead of the reached set of the primary analysis it would be better to return the reached set of the invariant generator, which is however not possible with the current available algorithms.*

Another drawback of the current invariant generation is the encoding of the invariants. According to the `InvariantSupplier` interface, an invariant is always a `BooleanFormula` [27]. This restricts the use of the invariant generator to analyses based on SMT formulas and more importantly, the formulas need to be encoded in the same way in both analyses, otherwise they cannot be combined. As an example, it is sufficient to consider an analysis that works with bit-precise SMT formulas, and an invariant generator that only approximates values using unbounded integers. Even if the naming of the variables in the formulas generated by the invariant generator is equal to the one of the primary analysis, due to the different types, the invariants are unusable. While this is a quite obvious requirement, there are also some hidden pitfalls, especially when it comes to pointer aliasing. In the following two sections, we will introduce our conceptual additions to overcome all mentioned problems, and also show our implementation of these additions.

[27] *A BooleanFormula is always an SMT formula, from our SMT backend JavaSMT, cf. `github.com/sosy-lab/java-smt`*

4.2 REACHED SET-BASED DATA EXCHANGE BETWEEN ANALYSES

As mentioned in the last section, due to the return type of the interface `InvariantSupplier`, using auxiliary invariants in CPAchecker is strongly tied to SMT-based analyses. There, each analysis that is supposed to use invariants generated with an `InvariantGenerator` implementation needs to know internal information about the encoding of the formulas to be able to use the invariants correctly.

By removing the `InvariantSupplier` completely, and instead returning the generated reached set when `get` is called on an instance of `InvariantGenerator`, we solve the problem of invariants being only usable within SMT-based analyses [28]. By retrieving all states for a certain location from the reached set, each consumer can then create the invariant in any encoding — SMT-based or not — individually.

[28] *For better encapsulation the return type of `get` will not be a reached set but a wrapper around one or more reached sets. More information on this can be found in Section 4.4.*

While in principle this solves all encoding related problems, and makes invariants usable for all analyses in CPAchecker, the handling of the invariant encoding was just moved to another location. Before our changes, implementations of the `InvariantSupplier` interface had to take care of the encoding such that it matches the encoding of the Predicate CPA [29]. Now, because we do not have the invariant as a `BooleanFormula`, but instead we have the reached set, we need to do the transformation ourselves in an appropriate place. For that reason, we added the `FormulaInvariantSupplier`, a wrapper class for reached sets, which computes the invariant depending on given parameters such as the location or the information about pointer aliasing.

[29] *The Predicate CPA was the only analysis which used invariants in combination with k-induction.*

The next section shows a further generalization of asynchronous invariant generation in CPAchecker that is also based on analyses exchanging reached sets. Synchronous invariant generation — before the consuming analysis is run — is also possible. Therefore we extend the sequential combination of analyses (cf. Section 2.3.3) such that the reached set of an analysis that was run prior to another analysis can be used in the later analysis.

4.3 PARALLEL ANALYSES

Besides the encoding and the limitation of invariants to having the type `BooleanFormula`, another problem with the implementation of the invariant generation in CPAchecker is that if an invariant generator proves safety we could potentially use its result as a shortcut, instead of the result of the primary analysis. However from inside the CPA

in a `CPAAlgorithm` it is not possible to return something else than the reached set used by this CPA. Therefore we create an algorithm that has the following abilities:

1. It wraps several analyses that will be executed in parallel.
2. It allows distributing *finished* reached sets[30] to other, running, analyses.
3. It takes the first *valid* reached set[31] of its component analyses, returns it, and aborts all other analyses, because their results are no longer important.

[30] *Usually each analysis results in one reached set. By using a variation of Algorithm 3 an analysis could return more reached sets, with increasing precision.*

[31] *The validity of a reached set depends on the soundness of the analysis, e. g., for soundness the waitlist has to be empty if no target state is in the reached set, because if the waitlist is not empty we have not fully explored the state space, and can therefore not be sure that further exploration does not result in finding a reachable specification violation.*

Item 1 and 2 provide the basic feature of having an asynchronously running analysis, like it exists in `CPAInvariantGenerator`. Out of the finished reached sets, invariants can be computed. The feature of having continuously-refined invariants implemented within the class `AutoAdjustingInvariantGenerator` is also available in the new `ParallelAlgorithm` (cf. Algorithm 5). For ease of presentation, we assume that all analyses are sound and precise and therefore no extra handling is required. In the implementation, soundness (no target states missed) and precision (target states are really target states) of an analysis are taken into consideration. If some conditions are not matching, e. g., the analysis is unsound but no specification violation was found, we ignore the result of this analysis and use the result of another of the concurrently running analyses if available.

Instead of invariants, the reached sets out of which the invariants can be extracted are given by the variable `aggregated`. The invariants can be retrieved orthogonally to `getCurrentlyKnownInvariants` from Algorithm 3. While in the `CPAAlgorithm` no specific retrieval of reached sets is specified, we consider that calling this method will be up to the CPAs and can, for example, be done in the transfer relation. The method `containsTargetState` is the same as for Algorithm 3. The methods `cancel_other_threads` and `join` are used for canceling concurrently running analyses, and for waiting on each concurrently running analysis to stop.

Overall, the refactoring of extracting the asynchronous invariant generation into the more generic `ParallelAlgorithm` has two major benefits. First, if safety can be proved with the invariant generator, we can use it without the need of incomplete reached sets (cf. Section 4.1). Second, the combination of several analyses (all of them can potentially be used as invariant generators) in parallel was not possible before, but is now. This can, e. g., be used like a sequential combination of analyses, with the difference that these analyses are run in parallel.

Input: a list L of quadruples with the following components:

1. a configurable program analysis with dynamic precision adjustment $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$,
2. a set $R_0 \subseteq (E \times \Pi)$ of abstract states with precision,
3. 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,
4. and a boolean flag that shows if the analysis should be continuously-refined

Output: the set reached and the set waitlist
Variables: a set reached of elements of $E \times \Pi$,
a set waitlist of elements of $E \times \Pi$,
thread-local versions of both variables,
a thread-safe set aggregated of sets reached for communication between analyses

```
1  aggregated := {}
2  For each (CPA, R_0, W_0, refined) ∈ L do in parallel
3      if refined then
4          Loop
5              reached := CPAAlgorithm(CPA, R_0, W_0)
6              R_0 := {(e, RefinePrec(π))|(e, π) ∈ R_0}
7              W_0 := {(e, RefinePrec(π))|(e, π) ∈ W_0}
8              if waitlist ⟺ ∅ then
9                  aggregated := aggregated ∪ {reached_thread}
10             if ¬containsTargetState(reached_thread) then
11                 reached := reached_thread
12                 waitlist := waitlist_thread
13                 cancel_other_threads()
14                 break
15     else
16         reached_thread := CPAAlgorithm(CPA, R_0, W_0)
17         if waitlist ⟺ ∅ then
18             aggregated := aggregated ∪ {reached_thread}
19         reached := reached_thread
20         waitlist := waitlist_thread
21         cancel_other_threads()
22 join()
23 return (reached, waitlist)
```

Algorithm 5: `ParallelAlgorithm`

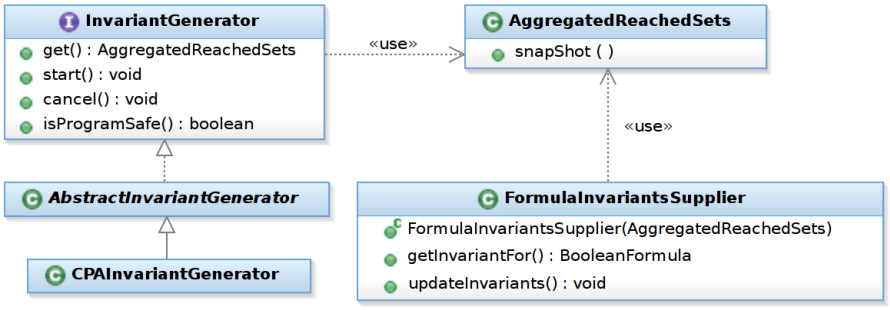

Figure 3: Invariant generation for SMT-based analyses (new)

4.4 ARCHITECTURE AFTER THIS THESIS

With the new concepts introduced in the last sections, some changes to the software architecture of CPACHECKER become necessary. First, we need to implement the `ParallelAlgorithm` [32] and a thread-safe means for exchanging reached sets, called `AggregatedReachedSets` in this thesis. Additionally, we have added the possibility of passing an `AggregatedReachedSets` object, from one analysis to the next, in a sequential combination of analyses. Due to our changes the asynchronous invariant generation can now be done as a parallel analysis. Therefore we can remove this functionality and the classes that are responsible for the (automatic) adjustment of the analysis from the former `InvariantGenerator` implementation. The only feature which is still available via the `CPAInvariantGenerator` is running an analysis sequentially. This feature is different to a sequential combination of analysis because the `CPAInvariantGenerator` can be run on the fly inside another analysis. This is, for example, important for path invariants later on.

[32] *See Figure 1 for the alignment of this algorithm in* CPACHECKER.

Figure 3 shows how invariants for SMT-based analyses can be computed. The structural alignment of the `ParallelAlgorithm` can be found in Figure 1. In addition to the functionality shown in Algorithm 5, the implementation is able to exclude certain reached sets from the `AggregatedReachedSets`, such that they are only used as direct return values of the `ParallelAlgorithm` if applicable. The same applies to the sequential combination of analyses, where one can configure that later analyses use reached sets of the earlier executed analyses.

5

AUGMENTING PREDICATE ANALYSIS WITH INVARIANTS

Before this master's thesis, invariants were used in CPACHECKER only for k-induction and path invariants [33]. The implementation of path invariants was not complete and had issues related to the problems with the `CPAInvariantGenerator` and `InvariantSupplier` as stated in Section 4.1. In the following, we describe the various options for utilizing invariants for enhancing the Predicate CPA. First, we focus on locations where invariants can be added, then, new approaches to invariant generation are described. In the last section of this chapter, we introduce a generalized handling of invariants for the Predicate CPA.

[33] *Path invariants are a Predicate CPA specific feature.*

5.1 INVARIANT INJECTION STRATEGIES

In Section 2.3.5 the Predicate CPA was introduced. Its states consist of two formulas, a path formula φ and an abstraction formula ψ. Additionally, each state has a precision π, which consists of predicates that are used to compute the abstraction formula at abstraction locations specified by the `blk` operator. The three components, precision, path formula, and abstraction formula, are potential candidates for adding invariants. The following sections provide more detailed insights on the advantages and drawbacks of adding (potentially-) invariant formulas [34] to each of these parts.

[34] *Differences in formula encoding (mainly due to pointers, or variables not tracked in the consumer analysis) may lead to having potentially invariant formulas instead of definite invariants.*

5.1.1 *Using Invariants as Precision Increment*

The precision π of the Predicate CPA contains predicates that are used for computing the abstraction formula during precision adjustment. During a normal analysis, the precision is initially empty, and predicates are added in the refinement step of the CEGAR algorithm (cf. Algorithm 2). The new predicates, called precision increment, are usually computed by interpolation, but they can also be generated in other ways, for example by heuristically mining them from the CFA [35]. In this approach, we add invariants as precision increment. This is the safest way of using potentially-invariant formulas compared to the other

[35] *Assume statements could, e. g., be used as predicates.*

two injection options, as it is not important that the added formula actually is an invariant for the running analysis. When computing the new abstraction formula $\psi' = (\varphi \wedge \psi)^\pi$ with the precision π we can use arbitrary predicates p_i as precision objects[36]. These predicates are only part of the new abstraction formula if $\varphi \wedge \psi \wedge \bigwedge_{p_i \in \pi}(p_i \iff v_i)$ has satisfying assignments containing the predicates. Thus, adding invalid predicates to the precision has no negative effect on the accuracy of the analysis. Overall, adding invariants as predicates to the precision is safe, but comes with the drawback that performance may suffer, as the all-sat computation becomes more expensive with growing size of the precision. The performance drawback depends on the used invariants: if they are strong enough to refute the counterexample found with CEGAR without further predicates, then the performance should not be differing much from only using interpolation, as the size of the precision does not increase more than it would be increasing by doing interpolation.

[36] *We do also need a propositional variable v_i for each p_i, cf. Section 2.3.5.*

5.1.2 *Appending Invariants to the Path Formula*

The path formula φ is computed in the transfer relation of the Predicate CPA by using the strongest post operator $\mathtt{SP}_{op}$ such that the successor abstract state $e' = (\psi', \varphi')$ for an abstract state $e = (\psi, \varphi)$ and a CFA edge $g = (l, op, l')$ is given by $\varphi' = \mathtt{SP}_{op}(\varphi)$ and $\psi' = \psi$. During precision adjustment, if a block abstraction should be done, the path formula is reset to TRUE and the abstraction formula is computed. Conjoining an invariant inv to the path formula before computing the abstraction results in the formula $\psi' = (\varphi \wedge \psi \wedge inv)^\pi$ due to the associativity of the logical and. Due to the immediate abstraction the invariant has only a very small chance of affecting the analysis. Therefore, we also conjoin the invariant to the new path formula after the abstraction[37]. Thus, instead of resetting the path formula to TRUE we reset it to the invariant in this case. This implies that the conjoined formula needs to be an invariant, a potential invariant is not enough, as otherwise we create an incorrect formula. An obviously incorrect example is to add FALSE as invariant, which makes the whole formula FALSE, and therefore leads to wrong results for the analysis. In contrast to that, adding TRUE, which is always an invariant, does not change the abstraction computation, just as intended. As stated in the introduction of this section, we do not have drastically wrong invariants such as FALSE, but there may be some encoding issues[38]. In the evaluation, we will see that in most cases, it is safe to use the invariants generated by CPACHECKER for appending it to the path formula, too. Additionally, this approach doesn't have the performance drawback that may result from incrementing the precision with invariants.

[37] *The necessity of this step is explained later on in the example in Section 5.1.4.*

[38] *Path invariants are also not applicable here, this is discussed in Section 5.2.1.*

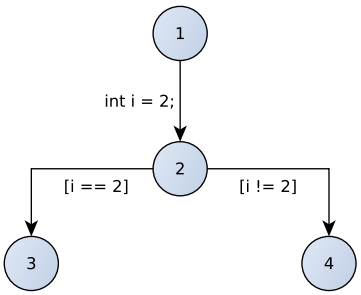

Figure 4: A CFA for illustrating the usage of invariants

5.1.3 *Appending Invariants to the Abstraction Formula*

Lastly, we can append computed invariants to the abstraction formula directly $\psi' = (\varphi \wedge \psi)^{\pi} \wedge inv$. In this approach, it is critical that the conjoined formula is an invariant. There is no filtering of invalid predicates, and also no abstraction of the path formula that abstract from potentially invalid formulas such that we still have valid formulas afterwards [39]. If we are sure that we have invariants, this is the approach that is the least detrimental for performance, as neither the size of the path formula nor the size of the precision increases, both of which might have negative impact on the performance of the satisfiability query used for the abstraction.

[39] *Adding non-invariant formulas to the path formula is still unsound, but might not be noticed due to the described effect.*

5.1.4 *Combining Invariant Use-Cases*

The approaches on utilizing invariants in the Predicate CPA introduced in the last three sections can also be combined. For example, appending invariants to the path formula is not helpful if the precision does not contain the necessary predicates. By adding an invariant to the path formula and the precision, we can therefore increase the accuracy of the analysis. In the following we will show the differences on an example.

Consider the control flow shown in Figure 4, the global precision $\pi = \{i < 10\}$ and the location invariant $i = 2$ for location 2. `blk` considers locations in front of conditions as being abstraction locations. We start with the state $e_0 = (\text{TRUE}, \text{TRUE})$ and compute the strongest postcondition for the path formula for the transition of location 1 to 2, the successor is then given by $e_1 = (\text{TRUE}, i = 2)$. Location 2 is right before two assume edges, and according to `blk` we treat it as the end of a block. Thus, we have to compute the abstraction before continuing. Now we have several options: we can refrain from using invariants at all (**No Inv**), we can add the invariants to the precision (**Prec**), we

Table 1: Differences in using invariants at different locations in the Predicate CPA

Strategy	New Abstract State	Possible Transitions
No Inv	$(i < 10, \quad \text{TRUE})$	$2 \to 3, 2 \to 4$
Prec	$(i = 2 \wedge i < 10, \quad \text{TRUE})$	$2 \to 3$
PF	$(i < 10, \quad i = 2)$	$2 \to 3$
AF	$(i < 10 \wedge i = 2, \quad \text{TRUE})$	$2 \to 3$
Prec + PF	$(i = 2 \wedge i < 10, \quad i = 2)$	$2 \to 3$
Prec + AF	$(i = 2 \wedge i < 10 \wedge i = 2, \quad \text{TRUE})$	$2 \to 3$
PF + AF	$(i < 10 \wedge i = 2, \quad i = 2)$	$2 \to 3$
Prec + PF + AF	$(i = 2 \wedge i < 10 \wedge i = 2, \quad i = 2)$	$2 \to 3$

can append them to the path or abstraction formulas (**PF**, **AF**), or any combination of these.

Table 1 shows the different strategies, the path formula and the abstraction formula *after* the abstraction, and the feasible transitions in the CFA. By simplifying the given formulas, some predicates could be omitted, but this is an expensive task for the SMT solvers, so we do not simplify them here to show the potential redundancy caused by using invariants. What can be seen is that any of the previously described invariant-usage approaches is sufficient to prevent the analysis from taking an invalid transition. For **PF**, this is only the case because we use the invariant as the new path formula instead of TRUE. Otherwise, the information that $i = 2$ would be lost due to the coarse precision, and the analysis would behave as if no invariants are used. Additionally, it does not make sense to use combinations of one of these approaches with **AF**, because this always results in duplicate clauses in the new formula. In case of **PF + AF** this is not obvious, because the duplicate formula will only come at the next abstraction, when the old abstraction already contains the invariant, and the path formula which gets conjoined to the abstraction formula does also contain it. In contrast to **PF + AF**, the combination of adding invariants to the precision and conjoining them to the path formula makes sense, otherwise one cannot be sure that the invariant conjoined to the path formula provides any benefit, because the precision might be too coarse. Using this combination is close to **AF** as it is very likely that the invariant predicate from the precision holds and is therefore used in the abstraction formula afterwards, this can also be seen in Table 1, the only difference is that for $Prec + PF$ the new path formula starts with the invariant instead of TRUE.

Overall, the approaches **Prec**, **PF**, **Prec + PF** and **AF** seem to be most promising, where **PF** should be worse than **Prec + PF** due to

the issues discussed in the last paragraph. **Prec** is the best option when only potentially-invariant formulas are used. For the other approaches, we need to be sure that we have real invariants. In the evaluation, we will have a look at all possible combinations of invariant usage strategies and compare their performance.

5.2 NEW INVARIANT GENERATION APPROACHES

In Chapter 4 the generalization of asynchronous invariant generation was introduced. In this section we explain all invariant generation approaches that are used later on in the evaluation. All of them are implemented in CPACHECKER and do not rely on external invariant generators. The approaches are divided into three parts: first we focus on invariants computed out of reached sets. Second we move on to sharing precisions. The third part consists of lightweight invariant-generation heuristics that are tied to the usage of the Predicate CPA.

5.2.1 *Sharing Finished Reached Sets*

By generalizing the idea of continuously-refined asynchronous invariant generation in Section 4.4, we now have the possibility to use finished reached sets not only from `CPAInvariantGenerator` for invariant generation. For example, we can have a sequential combination of analyses where the first analysis is very coarse. Due to infeasible counterexamples, we can not use the result of this analysis, but we can use its reached set for generating invariants for the next analysis, such that some of the computational effort of the first analysis was not completely wasted. By combining analyses with different strengths this way, we might be able to prove safety of programs where it would not be provable otherwise. With parallel analyses, reached sets can only be exchanged in a meaningful way, if one of the parallel analyses is continuously refined [40], such that we have a quickly terminating analysis whose reached set can be provided to the other running analyses. Both approaches will be analyzed exhaustively in the evaluation.

A special case, neither completely sequential nor parallel, are path invariants. They are computed sequentially, not before the consumer analysis is executed, but in between instead. Path invariants [BHMR07] for the Predicate CPA were introduced in CPACHECKER as part of a seminar work. They suffered from encoding problems described in Section 4.1 but this was not recognized, because path invariants can only be used as precision increment (cf. Section 2.4. For this thesis, path invariants were rewritten and integrated with the

[40] *Even running a fast analysis in parallel, which is not continuously refined, does not make much sense. This analysis could then simply be used in a sequential combination and provide the computed reached set already at the beginning to the consumer analysis.*

other invariant generation approaches. For generation of path invariants we run an analysis restricted to the counterexample path inside the current analysis via the `CPAInvariantGenerator`. Afterwards, the invariants are retrieved location-wise and used as precision increment.

While sharing reached sets is a generic approach that can be used by any analysis, some additional code is necessary such that an other analysis can take advantage of the given reached sets. For the Predicate CPA it is important that the invariants are SMT formulas. Creating an SMT formula out of a state in a reached set is not implemented for all CPAs available in CPAchecker. Thus we are restricted to the analyses working on states we can use, which are those, implementing the interface `FormulaReportingState` in CPAchecker. By using the `FormulaInvariantsSupplier` introduced in Section 4.2 we can then obtain invariants out of reached sets containing states of this kind.

5.2.2 *Sharing Precisions*

For sequential combinations of analyses, we do not only have the possibility to use the reached set of the earlier analyses in the later ones, but we can also dump the precision of the earlier analysis and use this precision in later analyses. This is, however, bound to certain CPAs as the precision is a CPA-specific object. Still, a fast primary analysis discovering some initial predicates for a later, slower but more precise, analysis could make sense. This approach does not rely on having real invariants, but it is related to adding invariants to the precision as described in Section 5.1.1.

5.2.3 *Lightweight Heuristics*

In this section we focus on invariant generation via heuristics. These heuristics are not guaranteed to find invariants, so their applicability greatly depends on the computational overhead they have in case no invariants are found. For this thesis, three different heuristics were invented which are described in the following paragraphs.

CHECKING INTERPOLANTS WITH k-INDUCTION Slicing an infeasible counterexample path into distinct — still infeasible — path prefixes and then selecting the prefix which should be used for refinement is a technique to guide the refinement [BLW15]. We do not select one prefix out of the computed prefixes, but instead we take all

of them and check each on 1-inductivity with *k*-induction. The nondeterminism of the SMT solver allows us to compute even more prefixes by creating interpolants more often for the same formula. The amount of unique interpolants found depends on the amount of possible solutions, so we chose to have three interpolation runs for the same formula as default. This number can be changed via a configuration option. The invariants discovered this way, can then be used to either increment the precision or conjoining them to the path formula or to the abstraction formula.

INDUCTIVE WEAKENING OF PATH FORMULAS Formula slicing is a technique for finding an invariant by weakening a given loop precondition based on the effects of the loop transitions [KM16]. The weakening process is guided by counterexamples-to-induction by an SMT solver. This approach is implemented in CPACHECKER. A complete analysis configuration is available with the name *formula-slicing*. We use this technique as a blackbox and provide the necessary input: the path formula before the loop start and the loop transitions. The result when using this blackbox is an invariant which we can use. In the worst case, the invariant is simply TRUE, so besides additional runtime we have no bad side-effects and the generated invariants can be used to either increment the precision, or conjoining them to the path or abstraction formula.

CHECKING CONJUNCTS OF PATH FORMULAS ON INDUCTIVITY Equally to weakening the path formula by removing clauses until the formula is inductive, we have added a heuristic that at first transforms the path formula into conjunctive normal form (CNF) [41] and then splits the formula into its conjuncts. The conjuncts are separately checked on 1-inductivity with *k*-induction. The invariants discovered this way, can then be used to either increment the precision, or appending them to the path or abstraction formula.

[41] *Our CNF conversion tool does support to not create a full CNF but to only have it on higher levels such that the exponential explosion of this transformation can be omitted.*

5.3 GENERALIZED INVARIANTS HANDLING IN THE PREDICATE CPA

In the last sections several different invariant generation and usage strategies for the Predicate CPA were introduced. To simplify usage and provide a clean interface, all invariant generation approaches are centralized in the class `PredicateCPAInvariantsManager` (cf. Figure 5).

This class aims at providing all necessary features and hiding all invariant-generation related details. It does also handle the computa-

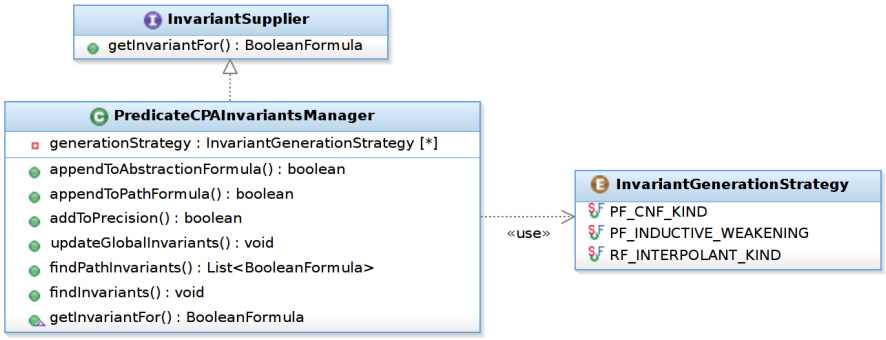

Figure 5: Managing invariants in the Predicate CPA

tion of invariants from reached sets of other analyses. The generation of invariants is strictly separated from the retrieval of invariants in order to increase the performance. In earlier implementations, invariants for a certain location were generated lazily as soon as they were requested. However, this is not possible for all invariant generation strategies we have. Additionally, when considering conjoining invariants to the path formula or the abstraction formula during precision adjustment, this happens very often, and thus takes a lot of time [42].

[42] *Abstractions are computed as indicated by* `blk`*, this is usually much more often than, e. g., refinements are computed.*

Our solution is to switch from the lazy generation approach to a more eager one: Now invariants are computed during refinement. The reasons for this solution are:

- the usually small amount of refinements, leading to few invariant generations but also guiding invariant generation to the important locations of the program [43], and
- the availability of information necessary for invariant computation, for example path invariants can only be computed during refinement as they need an infeasible counterexample path and information about the contained loops.

[43] *Computing invariants for program locations not leading to an error takes time that does not need to be spent.*

The following two sections provide detailed information about invariant generation and its usage.

5.3.1 *Invariant Generation*

Invariant generation in the Predicate CPA is split into three parts. First, we have the invariants computed out of reached sets of other, concurrently or sequentially running, analyses (we call them global invariants here), and at second, we have the invariants computed by the heuristics mentioned in the previous sections. The third part are

path invariants which are handled separately from the other invariant-generation approaches as their invariants cannot be used for conjoining them to the path or abstraction formula. The methods described in the following paragraph can also be seen in Figure 5.

Global invariants do not need further computation besides taking the reached sets and conjoining all states per location. Thus, updating them is easy and can be done by calling `updateGlobalInvariants`. This method is necessary to avoid changes to the global invariants between several invariant retrievals, for example, when retrieving invariants for adding them to precisions along an infeasible counterexample path, we want the invariants to come from the same reached set(s), and not from different ones. Due to the probably concurrently added new reached sets we need to decouple the updates on stored reached sets in the `AggregatedReachedSets` object from the reached sets used for invariant generation.

For locally computing invariants with one or more of the mentioned heuristics, the method `findInvariants` has to be called. There are several configuration options for this:

- The heuristics that should be used can be specified as a list, with the option *cpa.predicate.invariants.generationStrategy*.
- The heuristics in the list are executed in the given order, either until a heuristic succeeds in generating an invariant, or if all given heuristics should be used depending on the configuration option *cpa.predicate.invariants.useAllStrategies*.
- A time limit for invariant generation can be given with the option *cpa.predicate.invariants.timeForInvariantGeneration*.
- Besides the mentioned options there may also be separate options for each of the heuristics such as the analysis that should be used for the generation of path invariants.

Invariants generated in this way are cached for later usage, subsequent calls of `findInvariants` resulting in different invariants do not replace earlier results but are conjoined to them.

In contrast to that, path invariants can only be generated and retrieved at once, with the method `findPathInvariants`. Because they only hold for the specific given path, they are not cached but can only be used directly for the path they were generated for.

5.3.2 *Invariant Retrieval*

Retrieving invariants is done via the method `getInvariantFor`, receiving a location and the necessary information about pointer aliasing[44] as input. Depending on the configuration, invariants are retrieved from other reached sets (if available) and conjoined to the locally computed invariants by a heuristic. Calling `getInvariantFor` several times in a row for the same location and with the same pointer aliasing information is guaranteed to return the same invariants if none of the invariant-generation methods introduced in the last section were called in the meantime. For also having the information about the usage strategies available and configurable at one location in the Predicate CPA and not spread over several classes, we added the methods `appendToAbstractionFormula`, `addToPrecision`, and `appendToPathFormula` which indicate if invariants should be used for the given purpose.

[44] *Variables that are aliased by pointers are encoded in a special way, this encoding has to be added to the generated invariants.*

Part III

EVALUATION AND CONCLUSION

Within the next three chapters, we provide an exhaustive evaluation of different invariant generation strategies, showing their advantages and disadvantages. Furthermore, we give information about the problems we encountered. Lastly, we conclude the thesis by summarizing what we have accomplished, and giving and outlook on possible improvements for the future.

6 EVALUATION

In this chapter, the invariant generation and usage strategies introduced earlier will be evaluated regarding their performance. We use different kinds of programs for the evaluation and also several configurations to compare them to the assumed performance given in Section 5.1.4. We start by describing our evaluation setup and the used benchmarks, then the used CPACHECKER configurations are shown, and finally we take a look at the results.

6.1 EVALUATION ENVIRONMENT

The evaluation was performed on machines with two 2.6 GHz Octa Core CPUs (Intel E5-2650 v2) and 128 GB of RAM. The operating system is Ubuntu 16.04.1 LTS (64-bit) with a Linux 4.4.0-34 kernel. For the Java support OpenJDK 1.8 is used. The CPACHECKER revision for the evaluation is 23 084 (*trunk*). [45] Each verification run was limited to 2 physical CPU cores, which corresponds to 4 virtual CPU cores due to hyper-threading. RAM was limited to 8 GB. The overall CPU time for single-analysis verification runs was limited to 300 s, the concurrent or sequentially combined analyses have an overall CPU time limit of 600 s. This time limit may then be further divided by CPACHECKER internally, e. g., for concurrent analyses we aim at having approximately 300 s for both of the analysis parts. The Java heap was set to 6 GB for all analyses. For each verification run the overall amount of CPU time [46] and memory usage is measured. The benchmark execution and overall resource measurements are done with the BenchExec framework. [47]

In all tables time consumption will be given in hours with three significant digits unless it is specified in another way. In tables we refer to all programs, including timeouts, unknowns and other errors as **all**, the **correct** programs are a subset of this where we talk about all correct programs of a single configuration. For comparison purposes we also add information about the **equal** verification runs for a set of configurations. These numbers refer to the runs being analyzed correctly with each of the configurations in the table.

[45] *The configurations using path invariants were executed with revision 23146* (pathInvariants-fix)*. This revision contains a fix which only affects path invariants, and due to time limitations all benchmarks could not be rerun with this revision. The fix is also available in* trunk*.*

[46] *This time measure refers to the CPU time of the whole verification run, including all threads.*

[47] *More information on BenchExec can be found at* `github.com/sosy-lab/benchexec`*.*

6.2 BENCHMARK PROGRAMS

The benchmark programs we used are taken from the SV-Comp 2016.[48] We excluded the categories containing verification problems regarding memory safety, floats, termination, and concurrency. They are only analyzable with specific configurations of CPACHECKER which we do not use, or cannot be analyzed at all at the moment. Furthermore we did only use the program files of the other categories that needed at least 1 refinement while using our baseline configuration introduced in the next section. Tasks that can be solved without refinements will never be affected by our experiments, as invariants are only generated during refinements, so they are filtered out to present a clearer picture. In the end our benchmark set contains 3 488 verification tasks, where 2 413 are considered being *safe* and 1 075 are considered being *unsafe*.

[48] *The SV-COMP is a competition among automated software verifiers, more information can be found at* sv-comp.sosy-lab.org/2016/.

6.3 USED CONFIGURATIONS

The implemented invariant generation and usage approaches allow us to evaluate many different configurations. As the Invariants CPA produces bit-precise[49] formulas only, we can also only use bit-precise analyses. Additionally, all analyses using invariants are based on a predicate analysis. Furthermore some features of the predicate analysis that are not working correctly in combinations of the Predicate CPA and the Invariants CPA are switched off[50]. Due to using only bit-precise analyses we decided to use MATHSAT as SMT solver which is also the solver used by most CPACHECKER configurations for the SV-COMP. Other SMT solvers are not supporting bit vectors or they are not widely used and tested with the Predicate CPA.

[49] *Bit precise means that the formulas created with the SMT solver do not contain unbounded integers for integer variables, but instead fixed-size bit vectors are used.*

[50] *These features are an initial static refinement, which computes certain predicates out of the CFA, and that irrelevant variables, identified by another part of* CPACHECKER*, are ignored.*

For all configurations we have three baselines that are not using invariants, but have the same restrictions regarding unsupported features: **base300** and **base600** are predicate analyses with a time limit of 300 s, and 600 s respectively. **basePar** is a parallel combination of a predicate analysis and an analysis with the Invariants CPA — a portfolio analysis — where each of the analyses has approximately 300 s and overall they have 600 s together.

The configurations using invariants are divided into three categories which will be evaluated separately:

- First we have the lightweight heuristics which were introduced in Section 5.2.3, all of them can be used at any given invariant usage location or at a combination of them. The configurations

without specified invariant usage will be called as follows: **int-check** is the configuration which checks interpolants for inductivity, **weakening** is the configuration which weakens the path formula until it is an invariant, and **conj-check** is the configuration that transforms the path formula to a CNF and checks the conjuncts on inductivity.

Additionally to these heuristics we also have path invariants. They can only be added to the precision. Path invariants can be computed with every analysis having a state implementing `FormulaReportingState`. To show that this approach is not dependent on a specific analysis, we use in one configuration the Invariants CPA and in another configuration the Policy CPA [51] as invariant generation analyses. We will refer to these configurations as **path-inv** and **path-policy**.

[51] *The Policy CPA is based on local policy iteration [KMW16]. It also uses the Predicate CPA and the same formula encoding, such that we can exchange invariants easily.*

- Secondly we have the concurrent combination of the predicate analysis with an auxiliary-invariant generation analysis (cf. Section 5.2.1). The only analysis which can be used in a continuously-refined manner is, at the moment, an analysis with the Invariants CPA. All computed invariants can be used at any given invariant usage location or at a combination of them. These configurations will be prefixed with **async**.
- Lastly we have the sequential combination of analyses where results of earlier analyses are used for invariant generation in later analyses. As for both other categories, invariants generated with this approach can be used at any location. The name of these configurations will be prefixed with **seq**.

The different invariant usage strategies are forming the configuration name, where **-prec** means that invariants are added to the precision, **-path** means that invariants are conjoined to the path formula, and **-abs** means that invariants are conjoined to the abstraction formula. Combinations of these can also be used.

6.4 RESULTS

After explaining our evaluation environment, the used benchmark programs, and the set of configurations in the last sections, we take a look at the results of our evaluation process. This section is divided into three parts: first we focus on the configurations using lightweight invariants, such as path invariants or the inductive weakening of formulas; second we have a closer look at the configurations using parallel combinations of analyses for invariant generation and usage, and finally we look at the results of sequential combinations of analyses.

Table 2: Details on analyses using lightweight heuristics for generating auxiliary invariants and their baseline

	correct		wrong	Invariants (equal)			CPU time (h)		
	proof	alarm	alarm	time (h)	attempts	succ	all	correct	equal
base300	1 391	553	27				149	26.0	17.8
weakening-path	1 379	534	27	1.71	5 920	0	151	26.2	19.7
path-policy	1 337	529	27	3.52	3 950	1 498	161	31.4	25.7
int-check-prec	1 334	483	23	6.59	5 978	1 253	165	29.4	27.5
conj-check-path	1 384	543	27	0.674	5 920	0	147	26.5	19.0

The raw data for the tables and figures presented in this evaluation can be found at our supplementary web page at sosy-lab.org/research/msc/stieglmaier/. We do also show how our experiments can be reproduced and provide all necessary additional files.

Overall, the usage of lightweight heuristics, as well as the sequential combination of analyses, did not achieve a noticeable performance improvement compared to the baseline. The additional time taken for invariant generation is missing for the main analysis and furthermore, the invariant generation was often not even successful. In contrast to these results, the parallel combination of the Predicate CPA and the Invariants CPA lead to a huge performance boost. Compared to the single analysis baselines around 100 tasks more could be verified successfully and around 10 fewer false alarms were raised. More details and insights into all results are provided in the next sections.

6.4.1 *Lightweight Heuristics*

With lightweight heuristics we mean all approaches that can be computed on the fly and which should — compared to running more analyses in parallel or sequential combinations — take only short amounts of time. The configurations we describe here are path invariants, inductive weakening of path formulas, checking the invariance of interpolants and checking the invariance of conjuncts of the path formula. All configurations have a limit of 300 s overall CPU time, there is no extra time for the invariant generation.

In Table 2 the best configuration for each of the heuristics can be seen [52]. The columns show the number of correctly analyzed programs divided into found proofs and alarms. Additionally the wrong results are displayed. As only safe programs were erroneously treated as unsafe programs the other column was left out. The statistics about invariants show the **time** and the amount of invariant generation **attempts** as well as the amount of successful invariant generations (**succ**)

[52] *The values in the* Invariants *column are extracted from logfiles, so they are only available for the verification tasks not running into timeouts or experiencing other errors. Additionally they are measured by* CPAchecker *itself, so they are not as reliable as the CPU time which is measured by BenchExec.*

for all equal and correct verification runs. The last part of the table are the statistics about the CPU time.

The table shows that all lightweight invariant-generation approaches take too much time away from the main analysis, and furthermore, in this time not enough, or not the required invariants are found. The approaches of weakening path formulas or checking the conjuncts of path formulas transformed into a CNF were used about 6 000 times per configuration over all correctly analyzed programs, but not a single valid invariant could be generated. This can also be seen from the CPU time taken by the equal verification runs: The time for analyzing these programs increased approximately by the time needed for the invariant generation tries. Due to the higher time consumption also fewer programs could be analyzed in time.

Path invariants and checking interpolants on invariance lead to even worse results. While these configurations are able to generate invariants — and both add the invariants to the precision — they are taking even more time and therefore the performance suffers. For **path-policy** and **base300** the difference in the CPU time is over 4 h higher than the time for the invariant generation. This leads to the conclusion that the invariants have a negative impact on the performance, which could be the case for example, by adding predicates to the precision which force that loops have to be unrolled, an issue we wanted to overcome with these approaches [53]. More insights into the four heuristics are given in the following sections.

[53] *E. g., the predicate* `i = i + 1` *with* `i` *being a loop counter could cause this issue. An example where such formulas are found as interpolant, and the invariant is helpful follows in the section about the results with path invariants.*

Weakening of Path Formulas

Weakening path formulas up to the point where the remaining formula is an invariant did not work as expected. As can be seen in Table 2, this approach did not find any invariants. Without invariants all configurations we have tested are the same, as they only differ in where the invariants would be added. So besides minor changes to the results, which are caused by tasks that can be analyzed in approximately 300 s and therefore time out in some configurations but are successfully analyzed in other configurations, there is no difference. Upon further investigation we found several bugs in the usage and implementation of the reduced CNF conversion, they are fixed on later revisions of CPACHECKER than the evaluation was made on. An additional limitation to solvers that support quantifiers was necessary. Quantification is used for removing variables not having the most up to date SSA index in the process of converting a path formula to a reduced CNF. The combination of bitvectors and quantifiers is only possible with the SMT solver Z3. Unfortunately the integration of this

Table 3: Details on analysis using weakening or checking path formula conjuncts with Z3 instead of MATHSAT

	correct		wrong		Invariants (all)	
	proof	alarm	proof	alarm	attempts	succ
z3-base300	1 155	297	0	21		
z3-weakening-abs	1 047	249	5	25	7 802	5 523
z3-weakening-prec	965	250	0	20	8 442	6 026
z3-int-check-abs	1 111	254	0	18	6 976	1 648
z3-int-check-prec	1 093	249	0	21	7 117	1 792

solver in CPACHECKER is not perfect, and the results are not comparable to analyses with MATHSAT.

Table 3 shows some experimental results made with revision 23206 (*trunk*). Instead of MATHSAT, Z3 was used. This leads to a drastic performance decrease. By comparing **base300** with **z3-base300** we can see that 492 fewer tasks can be verified successfully. When using weakening of path formulas for invariant generation, the number of successfully analyzed tasks decreases even further, although the ratio of invariant generation attempts to successful invariant generations is higher than for the other lightweight heuristics.

Checking Conjuncts of Path Formulas on Invariance

This approach transforms a path formula to a reduced conjunctive normal form and checks the conjuncts on invariance with *k*-induction. As explained in 54 the conversion of formulas to reduced conjunctive normal forms does not work as expected with MATHSAT. Some experimental results with Z3 can be found in Table 3. While this approach is strictly better than weakening path formulas, it is still not able to correctly analyze as many tasks as **z3-base300** does.

The main difference of this approach and the weakening of path formulas is how the conjuncts are checked on invariance. Here we use *k*-induction as a separate analysis for finding 1- inductive invariants. In contrast, weakening uses counterexamples to remove conjuncts which cannot be part of the final invariant [KM16]. Both approaches can currently only be used with Z3 and therefore suffer from a worse performance than analyses using MATHSAT. Additionally it seems that generating invariants with these approaches has no beneficial influence on the analyses. What can be seen, is that **-abs** configurations perform better than **-prec** configurations, which is caused by the computational overhead of adding invariants to the precision. This observation can be made for all invariant generation approaches we have evaluated.

Table 4: Details on analyses using checking interpolants on invariance and their baseline

	correct		wrong		Invariants (equal)			CPU time (h)		
int-check-	proof	alarm	proof	alarm	time (h)	tries	succ	all	correct	equal
base300	1 391	553	0	27				149	26.0	17.8
abs	1 339	482	1	23	6.76	6 197	1 011	164	28.6	27.0
path	1 341	490	1	23	6.76	6 218	1 007	164	29.2	27.1
prec	1 334	483	0	23	7.09	6 455	1 272	165	29.4	27.9
prec-path	1 335	485	1	23	6.72	6 348	1 172	164	29.0	27.5
abs-path	1 342	489	1	23	6.80	6 229	1 011	164	29.2	27.1
prec-abs	1 333	481	1	23	6.70	6 293	1 162	165	28.5	27.4
prec-abs-path	1 336	486	1	23	6.69	6 364	1 174	165	29.2	27.5

Checking Interpolants on Invariance

Checking interpolants on invariance with k-induction is assumed to be a rather lightweight-invariant generation approach. Slicing infeasible counterexample paths into several infeasible prefixes and choosing one of them for interpolant computation [BLW15] is a technique that tries to guide the refinement such that the found predicates have a more positive impact on the analysis than choosing another infeasible prefix would have. We extend this approach to not only search for one infeasible prefix that is used for interpolant computation, but instead we compute interpolants for each of the infeasible prefixes and afterwards check the computed interpolants on 1-inductivity.

Table 4 shows the results for computing invariants with that strategy combined with all usage strategies we introduced earlier. All configurations using invariants are strictly worse than **base300**. Fewer verification tasks — safe and unsafe — could be analyzed successfully, and the overall CPU time increases from 149 h to over 160 h. When looking only at the equal and correct tasks, the difference is growing to almost 10 h, an increase in the time spent of over 50 %. Most of the additional time, about 7 h, is spent by trying to generate invariants, which is successful in approximately 1 out of 6 cases. The time for invariant generation is however not measured as CPU time but as wall time, such that the comparison of these times makes not much sense.

To take a closer look at the differences in time consumption, Table 5 shows the CPU time and wall time separately for the correct and equal analyzed verification tasks that either failed or succeeded to use invariants. It is surprising that the increase in CPU time is higher for the tasks where invariant generation was successful. Compared to the baseline about 40 % more time are needed for these tasks but only 13 % more time is needed for the tasks where invariant generation was not successful. When using the wall time instead of the CPU

Table 5: Drastic increase of CPU time for analyses succeeding in using invariants computed by checking interpolants

		invariant generation failed				invariant generation succeeded			
		base300	**prec-abs-pf**	**prec**	**abs**	**base300**	**prec-abs-pf**	**prec**	**abs**
CPU time (h)		13.9	20.7	20.7	20.4	3.92	6.85	7.16	6.54
inv time (h)		0	5.00	5.04	4.99	0	1.41	1.74	1.35
	-	13.9	15.7	15.7	15.4	3.92	5.44	5.42	5.19
increase (%)			13.0	13.0	10.8		38.8	38.3	32.4
Wall time (h)		9.19	14.0	14.0	13.7	2.06	3.42	3.68	3.19
inv time (h)		0	5.00	5.04	4.99	0	1.41	1.74	1.35
	-	9.19	9.00	8.99	8.75	2.06	2.02	1.94	1.84
decrease (%)			2.07	2.18	4.79		1.94	5.83	10.7

time for comparison, the numbers are changing, for unsuccessful invariant generation the time decreases by 2–4 % and for the successful invariant generation even from 2–10 %. Both comparisons lead to the conclusion that other threads are influencing our measurement, and in fact when executing the benchmark set limited to one virtual core, wall time and CPU time are equal and the time for generating invariants is still smaller then the difference in the measured times. Our research did not come to any conclusion where the additional time — accounting about 3 h in our experiments — could be spent. Invariant generation, as well as all other parts of CPAchecker, are run single-threaded, the SMT solver MathSAT is also running single-threaded and while profiling the application we did not find any additional threads being used. Also when looking at the ratio of wall and CPU time, it stays approximately the same for the tasks with failed invariant generation (about 50 %) and successful invariant generation (about 90 %), which means that there is no evidence for additional time in configurations with invariant generation in particular, but a part of the used CPU time is always spent differently, for example, for garbage collection or resource measurement.

When we look back at Table 4 we can see that some of the configurations have one unsafe verification task, `ldv-linux-3.0/usb_urb-drivers-input-misc-keyspan_remote.ko_false-unreach-call.cil.out.i.pp.i`, where the analyses concluded that this program is safe. This is a side-effect of using invariants. Without invariants the analysis of this task does not terminate, with invariants being added to either the path or the abstraction formula, the analysis terminates and reports a wrong result. The baseline is not able to analyze this program, even in 900 s. Therefore we do not know if the wrong result is caused by invalid invariants, a wrong usage of invariants or if the program cannot be analyzed correctly with the given **base300** configuration. In the

SV-COMP 2016 there were two tools that terminated in time, one of them (Blast) reported a specification violation, so we can be sure that the problem is not that this verification task has a wrong label.

Another remarkable point is that some of the tasks are running into a timeout because of the sliced prefix generation. This issue can be observed better with increasing amount of possible slices that need to be tested. The initial idea was to increase the number of abstraction states by changing the block operator `blk` (cf. Section 2.3.5) such that abstractions should be computed more often. The default configuration is that an abstraction is only computed at each occurrence of a loop head. We add that additionally, an abstraction is computed when control-flow meets. With this modification, the infeasible counterexamples consist of more abstraction states than before, which means that there are also more states that can be removed for creating different infeasible prefixes. But with this increased number of possibilities, the number of timeouts rises, for example, for **int-check-abs** from 1 563 to 2 394. At the same time **base300** has 1 414 timeouts. For all other configurations for generating invariants with this approach the numbers are comparable. The reason for the longer prefix generation times is, in our opinion, that removing certain formulas from the satisfiability check has an impact on the SMT solver, which is in turn not able to prove unsatisfiability. Formulas leading to such a behavior when removed could, e. g., be related to pointer-aliasing handling, because for that, many relations are introduced. By removing some relations, the possible state space grows and makes the unsatisfiability-check harder.

Overall checking interpolants on invariance with k-induction seems to be working for only a small set of verification tasks. For the other tasks, either the invariant generation takes too long, or the found invariants do not influence the analysis in the expected way. The idea to increase the amount of interpolants being checked by changing the behavior of the `blk` operator made the performance even worse. While more infeasible sliced prefixes do also mean more interpolants, and potentially a higher success rate in finding invariants, the additional time necessary for the prefix generation is just too high.

Path Invariants

As stated in Section 3.2 and Section 2.4, path invariants were already a part of the CPACHECKER framework before this master's thesis, but during this work we found an issue with the old implementation. The conversion of formulas did not consider different pointer encodings and thus lead to wrong formulas used as precision increment in the Predicate CPA. This was not recognized, because adding additional

Table 6: Details on analyses using path invariants for generating auxiliary invariants and their baseline

	correct		wrong	Invariants (equal)			CPU time (h)		
	proof	alarm	alarm	time (h)	tries	succ	all	correct	equal
base300	1 391	553	27				149	26.0	21.3
path-inv	1 327	519	27	2.36	4 719	1 428	162	31.0	30.5
path-policy	1 337	529	27	3.84	4 600	1 611	161	31.4	29.8
400s-inv	1 364	575	27				196	35.6	
400s-policy	1 371	576	27				196	34.7	

formulas to the precision — correct and incorrect ones — does not lead to wrong behavior, only the runtime increases with increasing size of the precision. In our earlier work we found analyses using path invariants generated by the Invariants CPA or the Policy CPA perform better than the baseline in terms of the number of correctly analyzed tasks. Path invariants improved the results by about 1.5 %[54]. When it comes to the time measured, analyses with path invariants took significantly longer (more than 10 %) than the baseline. This comes on the one hand from the time needed for the invariant generation, and on the other hand from the additional time needed for repeated refinements, if the generated invariants were not strong enough to refute the counterexample[55].

[54] *This is not comparable to our results as we use bit-precise analyses and the earlier evaluation used unbounded integers and rationals.*

[55] *This can be the case due to over-approximation while converting the formulas or because of the fact that the formula encoding was not correct, and therefore many formulas added to the precision could not be used afterwards.*

Our evaluation shows completely different results. The number of successfully analyzed tasks while using path invariants is about 4 % lower than **base300**. This is mainly caused by the additional time needed for invariant generation. When increasing the time limit to 400 s (**400s-inv**, **400s-policy**), only verification tasks where the invariant generation is successful have a changing outcome, such that the overall results become approximately equal to **base300**. In Table 6 one can see that while **path-policy** takes about 1.5 h more time for generating invariants compared to **path-inv**, it can successfully analyze 20 tasks more, and additionally the CPU time for equally and successfully analyzed tasks is approximately 0.8 h less than for **path-inv**. This behavior is less noticeable for the **400s** configurations but still the analysis using the Policy CPA performs better.

What is also interesting is that there are many tasks that can be correctly analyzed by **base300** in 15 s to 30 s, but are running into a timeout with **path-inv** and **path-policy**. The invariant generation in these cases is not the problem; instead the usage of the generated invariants leads to loop unrollings, and in turn, to more refinements than **base300**. The aim of path invariants was initially to prevent loop unrollings (and therefore refinements) but it seems that this is not working as expected with this invariant generation strategy. From Table 6

Table 7: A selection of tasks and their results with path invariants

file name	path-inv	path-policy
loop-acceleration/array_true-unreach-call3.i	✓	✗
loop-acceleration/functions_true-unreach-call1.i	✗	✓
loop-acceleration/nested_true-unreach-call1.i	✓	✗
loop-acceleration/simple_true-unreach-call1.i	✗	✓
loop-new/count_by_1_true-unreach-call.i	✓	✗
loop-new/count_by_1_variant_true-unreach-call.i	✓	✗
loop-new/count_by_nondet_true-unreach-call.i	✗	✓

```
1 int main() {
2   int i;
3   for (i = 0; i < 1000000; i++) ;
4   __VERIFIER_assert(i == 1000000);
5   return 0;
6 }
```

Listing 1: The source code of loop-new/count_by_1_true-unreach-call.i

we can see furthermore that the number of correctly analyzed safe programs decreases in a higher ratio than the number of correct alarms. For the **400s** analyses, the number of correct alarms is even higher than **base300**, while the number of correct proofs is still smaller than **base300**.

Both path invariant generation approaches yield better results than **base300** for the tasks in the loops category. The tasks in the loops category do not consist of many lines of code. Instead they have loops and some conditions that are complicated to track without having relations between variables. While **base300** times out on some of the tasks after 300 s, either **path-inv** or **path-policy** are able to successfully prove the safety of these programs (cf. Table 7). Furthermore, the safety of these tasks can be proved within 10 s. So the speedup due to the invariants is enormous. For all these tasks **base300** times out after many refinements, with invariants only a small amount of refinements (between one and five) are necessary. Most of the computed invariants are very simple, for example, for the task *loop-new/count_by_1_true-unreach-call.i* (cf. Listing 1) the computed invariant is that at the location of their `__VERIFIER_assert` call `i = 10000`. This is very helpful in contrast to the interpolants found by the SMT solvers, which force a loop unrolling in this case, because in each loop iteration the interpolant found is just that `i` equals the next higher number.

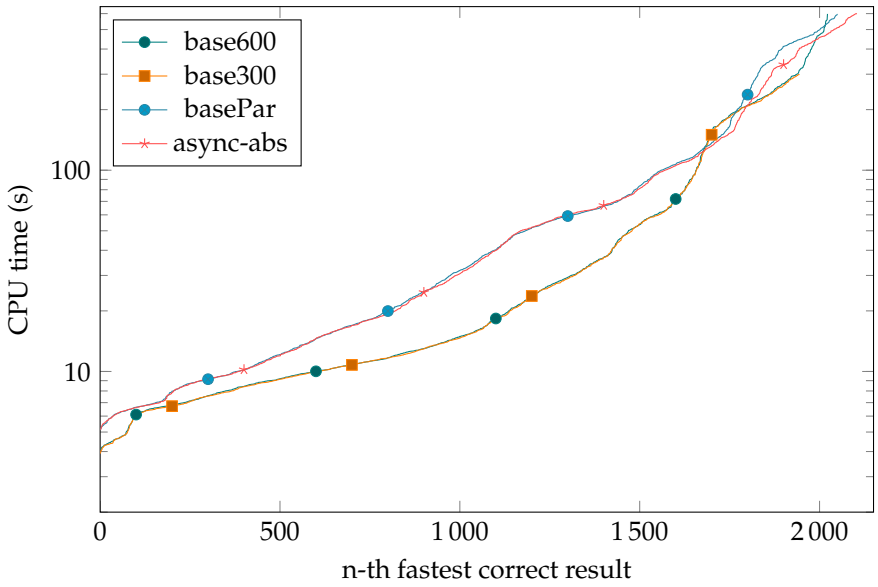

Figure 6: A quantile plot showing the best concurrent analysis and the three baselines

To sum up, we have three cases, first there are tasks where invariant generation is not successful and thus the additional time spent effectively slows down the analysis. Second, there are tasks where invariant generation is successful, but the found invariants slow down the analysis, for example by forcing a loop to be unrolled. Unfortunately this case is appearing to be more common than the last case where the auxiliary invariants speed up the analysis. The last case leads, in many cases, to results within seconds where otherwise five minutes are not enough to analyze the verification task. For a better performance it will therefore be necessary to classify the error traces and only compute path invariants for certain cases. Such conditions could, e. g., be that the loops in the infeasible counterexample path do not consist of more than `X` statements, or that the loop conditions and the loop iteration statements may only be simple increment or decrement operations. In general, taking the complexity of the loop body into account might help, but finding appropriate heuristics for that is future work.

6.4.2 *Parallel Analyses*

In this section we move away from trying to generate invariants but keeping the impact on the overall time of the analysis low, to using approximately half of the overall analysis time for invariant generation. By limiting the overall analysis time to 600 s and the CPU time to 300 s for both of the analysis threads we can achieve this. Tracking the

CPU time a thread needs is, however, not very precise in our case. On the one hand, threads started from within a thread are not counted towards this time, and on the other hand, additionally running threads, for example for handling the resource limits or also the Java garbage collector can also not be counted towards these limits. So the thread-wise CPU time limits are a best-effort approach to achieve a certain distribution.

As described in Section 6.3 we only have parallel configurations combining an analysis using the Predicate CPA and an analysis using the Invariants CPA. The Invariants CPA is configured to be continuously refined, and provides the intermediate finished reached sets to the Predicate CPA, which computes invariants from them. As baselines, we use all three configurations **base300**, **base600** and **basePar** to be able to draw more precise conclusions out of the results. Figure 6 gives an overview on the performance of all baselines and the best parallel analysis using invariants [56]. All parallel-analysis configurations will be discussed in more detail in the following paragraphs. Exact numbers for all configurations can be found in Table 8. As can be seen in Figure 6, **base300** has the same curve as **base600**, it just stops earlier. This is exactly what is expected as both configurations are equal, only **base600** has twice the amount of time. More interesting is that the number of successfully analyzed tasks increases by 78 from **base300** to **base600** but by instead using **basePar** we can further increase the number of successfully analyzed tasks by 28. This means that there are many programs quite hard to analyze with the **base300** or even **base600** which are easier for the analysis using the Invariants CPA. By generating invariants and using them at any possible location we can once again increase the number of successfully analyzed tasks. While all configurations using invariants perform strictly better than the baseline, **async-abs** is the best configuration we evaluated in this setting, with an increase of 54 more correctly analyzed tasks, compared to **basePar**.

[56] *All other parallel configurations using invariants are quite similar to* ***async-abs*** *and are excluded from Figure 6 for better visibility.*

Table 8 shows many interesting facts about this way of generating and using invariants. At first the overall CPU time of all analyses using parallel analyses is about 87 % higher than **base300**. This is not surprising as we configured the parallel analyses to be able to use at most 600 s, and there are two analyses running concurrently. Compared to **base600** the overall CPU time is only increasing by approximately 6 %. When only looking at the correctly and equally analyzed tasks, the parallel analyses consequently take about 90 % more CPU time than **base300** and **base600**. By looking at the wall time the picture changes. Over all tasks the wall time of **base600** is 240 h and the wall times of the parallel analyses are around 150 h, about 37 % lower. For the correctly and equally analyzed tasks the wall time for **base300**

Table 8: Details on all parallel analyses using invariants and their baselines

	correct		wrong		Main Succ	Wall time (h)		CPU time (h)	
async-	proof	alarm	proof	alarm	correct	all	equal	all	equal
base300	1 391	553	0	27	1 944	128	13.8	149	20.9
base600	1 434	588	0	27	2 022	240	13.9	262	21.1
basePar	1 509	541	0	18	1 109	152	15.6	281	39.9
abs	1 532	572	0	18	1 154	147	14.4	276	38.2
path	1 536	561	1	17	1 148	146	14.2	274	37.9
prec	1 526	549	0	18	1 108	149	15.3	279	39.6
prec-path	1 525	561	1	17	1 111	148	15.1	278	39.4
abs-path	1 528	568	1	18	1 148	146	14.4	275	38.4
prec-abs	1 526	557	0	18	1 110	149	15.2	279	39.4
prec-abs-path	1 531	551	1	18	1 106	148	15.0	278	39.5

and **base600** is shorter than any of the parallel analyses. It is noticeable that while the number of correctly analyzed tasks is increasing for all parallel configurations, the number of wrongly analyzed tasks decreases by 9, about 33 %. In the next paragraphs we will only use **basePar** for comparisons, because this baseline is closest to the configurations using invariants.

basePar is the slowest of all parallel configurations, meaning that the usage of invariants boosts the CPU and wall time of the analyses. The wall time is overall about 3 % higher and for the equal tasks about 8 % higher. For the CPU time it is overall about 2 % higher, and 4 % higher for the equally and correctly analyzed tasks. While the consumed time decreases when using invariants, the number of correctly analyzed tasks rises between 25 and 54, making the performance of the analyses strictly better than **basePar**.

By taking a closer look at the configurations using invariants we can see that the configurations where the invariants are conjoined to the path formula are analyzing one unsafe task wrongly and report that there is no specification violation. By digging deeper we found out that at some point, the conjunction of path formula and invariant became unsatisfiable, immediately leading to the wrong result. When comparing the path formula and the invariant at this point, one can see that the invariant assumes that a variable (a pointer) has the address zero, where it has another address in the path formula. This is no encoding issue in the way we thought about it, instead it has to do with how (aliased) pointers are handled in different CPAs. While the Predicate CPA handles such cases with uninterpreted functions and does not expect value assignments to such variables, there is no special handling in the Invariants CPA at all. The Invariants CPA uses a separate CPA for that, and the formulas generated by the Invariants CPA do unfortunately contain assumptions about pointers being 0. [57]

[57] *The full path formula, the invariant, and the interpolant for the program* `heap-manipulation/sll_to_dll_rev_false-unreach-call.i` *can be found on our supplementary web page.*

This leads to the unwanted behavior we observed here. That this problem leads to a different results only one time in over 3 400 verification tasks makes the problem even harder to find. A reason for this issue not having bad effects on the **-abs** configurations is that the interfering parts of the path formula are removed during abstraction due to the precision.

Apart from the wrongly analyzed task there are more differences, for example regarding the number of tasks where the result comes from the main analysis using the Predicate CPA and not from the additional analysis using the Invariants CPA. From 2 104 correctly analyzed tasks with **async-abs**, 1 154 results are reported by the main analysis, about 55 %. For all configurations where the invariants are appended to the precision, this ratio is about 53 %, meaning that the main analysis became slower such that more results are given by the analysis with the Invariants CPA. This can also be seen by the wall and CPU times, which are higher for all **-prec** configurations.

In Section 5.1.4 we made some assumptions about the performance of the different approaches, stating that **-abs**, **-path**, **-prec** and **-prec-path** look most promising. While **-prec** suffers from bad performance due to more necessary computations during the abstraction, we need to be sure that we have invariants (with correct encoding) to be able to safely use them with **-abs**. From Table 8 we can see that **async-abs** is the best configuration we have used, which is an indicator that our formula conversion works very well. Additionally we found out that the performance drawback of appending invariants to the precision is considerable. The configurations conjoining the invariants to the path formula are almost as good as **async-abs**, however, they have one wrongly analyzed unsafe result, making the conclusion about their validity impossible.

6.4.3 *Sequential Combination of Analyses*

From using some analyses concurrently we come to sequential combinations of analyses now. While for parallel analyses the advantage is that all concurrently running analyses can communicate via intermediate finished reached sets with each other, the drawback is that all analyses consume CPU time at the same moment although one analysis may perhaps be able to successfully analyze the program all by itself. With sequential combinations of analyses we try to use fast and coarse analyses at first and if they are not able to successfully analyze the program we pass their reached set on to the next analysis, which can save computation time due to the already found invariants.

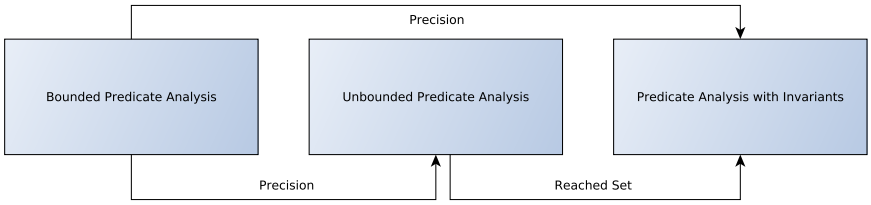

Figure 7: Overview over the sequential combinations of analyses and their information exchange

Our approach consists of two to three sequentially combined analyses. An overview can be seen in Figure 7, they are explained in detail in the following paragraphs.

1. The first analysis is always a bit-precise predicate analysis limited to a number of five loop iterations,[58] which means that all paths where more than five loop iterations are encountered are ignored in the further analysis. This technique is for example also used for BMC. The refinement of paths with possible specification is delayed until the full state space is explored. Then all paths are refined at once, and we terminate the analysis. The precision increment computed by the so called global refinement[59] is dumped for further usage with other analyses. This analysis can only be used for finding specification violations, as the loop bound prevents us from analyzing deeper parts of the state space; therefore we cannot draw conclusions about safety. This analysis is assumed to be very fast, but we still limited it to 100 s in our evaluation, as for some programs it takes longer than expected. If the time limit is reached, no precision is dumped and the next analysis is skipped, such that the additional time needed for unsuccessful tasks does not grow too much.

2. The second analysis is once again a bit-precise predicate analysis, but this configuration is unbounded. It uses the precision dumped by the first analysis and just explores the state space. This analysis does not use any refinement, its aim is just to provide a reached set we can use later on for invariant generation. As we do not use any refinement, this analysis cannot be used for finding bugs, instead it can prove safety if the precision computed by the first analysis is strong enough. This analysis is limited to 100 s.

3. The third analysis is a bit-precise predicate analysis which uses the reached set computed by the second analysis for computing invariants. The invariants can then be used for adding them to the precision or for conjoining them to the path or abstraction formulas. This analysis can do both, prove safety and find bugs.

[58] *Five loop iterations were chosen as a trade off between precision and a fast analysis.*

[59] *This refinement strategy was implemented by us especially for this invariant generation approach. The advantage is that by delaying the refinement we can explore the full state space for a given precision, which would not be possible with the existing refinement strategies.*

Table 9: Details on all sequential combinations of analyses using invariants and their baselines

	correct		wrong	∅ Analyses	Wall time (h)					
seq-	proof	alarm	alarm		Alg1	Alg2	Alg3	all	correct	equal
base300	1 391	553	27	1.00				128	17.9	14.5
base600	1 434	588	27	1.00				240	26.7	14.7
restart2	1 420	612	27	1.97	12.3	8.84		182	29.1	22.7
abs	1 415	557	27	2.38	12.3	3.35	8.73	201	32.3	25.9
path	1 416	547	28	2.38	12.3	3.39	8.65	200	30.9	25.9
prec	1 409	550	27	2.38	12.3	3.33	9.15	202	31.7	26.3
prec-path	1 409	557	28	2.38	12.3	3.40	8.89	201	31.7	26.1
abs-path	1 414	555	28	2.38	12.3	3.35	8.66	200	31.5	25.9
prec-abs	1 407	555	27	2.38	12.3	3.36	9.21	202	32.0	26.4
prec-abs-path	1 414	552	26	2.38	12.3	3.35	9.13	201	31.8	26.3

In our evaluation this analysis is limited to 300 s. Altogether we have two analyses for invariant generation and the main analysis, which uses the computed invariants afterwards.

As one approach to using invariants is adding them to the precision, we did also evaluate a configuration using only the analyses described in item 1 and item 3. The later analysis then uses the dumped precision directly and behaves like the baseline analyses apart from that. This has the advantage that no additional time for the state-space exploration of the second analysis is needed, and furthermore, there are no invariants that are changing the analysis during run time. This configuration will be called **restart2**.

Table 9 shows the results of evaluating the sequential invariant-generation approaches. While all configurations using invariants are slower than **base300** by about 40 % to 56 % they are strictly faster than **base600**. The speedup is about 25 % for **restart2** and a little lower for the other configurations. In general, **restart2** is the best configuration using invariants we have in this category: the additional time necessary for state-space exploration in the other configurations slows down the analysis too much. By looking at the successfully analyzed verification tasks, we can see that **restart2** performs overall a little better than **base600**, and **base300** is the worst configuration in this regard. The higher number of correctly found specification violations for **restart2** is caused by the first analysis of the sequential combination. For 14 fewer tasks safety was proved correctly by **restart2** compared to **base600**, which is mainly caused by the fact that the predicate analysis needs more than 300 s to analyze them (with and without invariants). Another cause that was already explained earlier is that some invariants explicitly cause some loops to be unrolled and therefore lead to timeouts where the baseline is able to analyze the task in time.

That the time for the first and second analyses — columns **Alg1** and **Alg2** — are always approximately the same is the case because the analyses are always the same. The difference is only in using the invariants in the third analyses. The times for the third analyses are also only differing marginally. Surprisingly, the time consumed in the first analysis is higher than the time in the subsequent analyses. To reduce this time, we could, for example, decrease the loop bounds to a value smaller than five. The average number of analyses used is therefore also equal: 2.38 means that many of the analysis either need two or all three analyses for analyzing the verification task. For **restart2** the average number of used analyses is 1.97, which means that in most cases the second analysis is required.

A closer look at the different invariant usage strategies shows the same picture as for the concurrently generated invariants evaluated in the last section. Apart from **restart2**, **seq-abs** is the best configuration using invariants. It is about 0.5 h (5 %) faster than **seq-prec**. **seq-path** is even faster than **seq-abs** but its results are not as good. **seq-prec-path**, which should be almost equal to **seq-abs** is once again too slow and also the results are worse than the results of **seq-path** alone.

Overall we can say that the combination of two analyses with different aims leads to better results in a shorter time than a single analysis does. **restart2** is clearly faster than **base600** with equal performance, and a bit slower than **base300** with much more correctly analyzed verification tasks. What can also be seen is that invariants should not be generated at all costs: in some cases it makes sense to use weaker assumptions, for example, a dumped precision for incrementing the precision of another analysis, instead of creating invariants out of this precision with a separate analysis.

6.5 CONCLUSION OF THE EVALUATION

Over all evaluated configurations, the concurrently computed invariants yield the best results, performing strictly better than the baseline. With **async-abs** being the best configuration overall. The sequential combinations of analyses were all better than **base300**, but most of them were not as good as **base600**. The lightweight invariant-generation approaches did not work as expected, resulting in a worse performance than the baseline. They still need more work to make them faster and the results more reliable. When looking at the different usage strategies, our assumptions about their performance made in Section 5.1.4 were true, with **-abs** being the best option if a correct invariant is given.

7 RESTRICTIONS AND CHALLENGES

While implementing the invariant generation and usage approaches for this thesis we encountered several difficulties. These are described in the following sections.

7.1 LARGE FORMULAS

Invariant generation with the approaches we have implemented is either based on path formulas or interpolants that are transformed, or on separate analyses where the invariants are generated out of the reached set of the separate analysis. All cases have in common that for most verification tasks the formulas that need to be handled are very large. Printing them would take several sheets of paper and therefore debugging is hard. Some problems can also only be observed for one or two tasks of the benchmark set, making them hard to find. This is for example the case for the differing pointer encoding described in Section 6.4.2. In general, encoding issues caused by transforming states of one CPA to formulas for the Predicate CPA were the main point we had to work on, while implementing invariant generation approaches based on other analyses.

7.2 EXTERNAL INVARIANT GENERATORS

The initial idea for this thesis was not to create many invariant generation approaches on our own, but to use existing invariant generators and supply the found invariants to our analysis. Unfortunately we did not find suitable invariant generators. All of the ones we found have individual drawbacks that make their usage for our purpose impossible. INVGEN and DAIKON are already mentioned in Section 3.4. Both look very promising, but the requirement of the INVGEN front end that only programs with exactly one function, and neither arrays, nor pointers are allowed makes this invariant generation unusable for us. The front end of DAIKON is able to handle all C programs we need, however it is just instrumenting the program and then tries to deduce likely invariants out of the values observed during run time. This means we cannot have any non-determinism in our programs.

We can only simulate non-determinism by either using fixed random numbers (and therefore limiting the state-space before the analysis, which is unsound), or by using a random number generator directly (this will most likely lead to non-termination, and is not sound either). Overall every attempt to integrate one of these invariant generators failed, so we had to implement some invariant generation approaches ourselves.

8 CONCLUSION

In this chapter we give a summary of this thesis. We briefly explain our conceptual additions and provide an overview over the results of our evaluation. Lastly we conclude our thesis with an outlook on how the usage and generation of invariants can be extend and improved in the future.

8.1 SUMMARY OF THIS THESIS

In this work we introduced a new algorithm for concurrent execution of several analyses and based on this we added the possibility of communication via reached sets in CPACHECKER. This was then used for computing invariants out of reached sets for usage with another analysis. Besides concurrent analysis we also implemented several lightweight invariant-generation approaches, either based on path formulas that are transformed to a CNF-like shape and then used to compute invariants by inductive weakening or simply checking the conjuncts for invariance, or they are based on interpolants that are checked for invariance. All the lightweight approaches did not yield the expected results, neither the path formula and interpolation-based approaches, nor path invariants. All of these approaches consumed to much time and therefore turned out not to be as lightweight as expected.

The configurations using invariants computed sequentially or concurrently are working better. The sequential configurations have a run time in between 300 s and 600 s thus being slower but much more precise than the baseline with 300 s and being faster and equally precise than the baseline with 600 s. For parallel analyses we measured a performance improvement of about 3 % compared to a portfolio analysis and about 8 % compared to a pure predicate analysis needing a comparable wall time.

The evaluation of the different invariant-usage strategies showed that our expectation was correct and conjoining invariants to the abstraction formula yields the best results. Adding invariants to the precision is the best option if we have formulas where we are not sure if they are invariant, but adding formulas to the precision also slows the analysis down.

8.2 FUTURE WORK

For the future there are several possibilities how this work can be extended. The main goal is to make the the inter-analysis communication more robust. With the current approach already small differences in the formula encoding will lead to hard-to-find errors. Furthermore we did only test each invariant generation approach on its own, by combining them we may be able to use synergies between them, e. g., the concurrent approaches are better in finding safety proofs whereas the sequential approaches find more bugs. Besides combining different approaches we could also try to make the existing approaches more intelligent, for example, such that invariants are only used if they are known to improve the analysis, and ignored in other cases. This could be achieved by classifying the parts of the invariant formula such that we try to find out which parts will, for example, lead to loops being unrolled. Another approach to improve invariants usage is to take, additionally to the location, the call-stack information into account. This way, invariants generated from reached sets contain less disjunctions and make the analysis more precise.

Overall our work is the base for future experiments with invariants in CPACHECKER. Its flexibility and modularity make future extensions easy. Several ways to enhance our work are described in the paragraph above.

BIBLIOGRAPHY

[AS06] M. Awedh and F. Somenzi. *Automatic invariant strengthening to prove properties in bounded model checking*. Proc. DAC, pages 1073–1076. ACM/IEEE, 2006.

[BDW15a] D. Beyer, M. Dangl, and P. Wendler. *Boosting k-Induction with Continuously-Refined Invariants*. Proc. CAV, LNCS 9206, pages 622–640. Springer, 2015.

[BDW15b] D. Beyer, M. Dangl, and P. Wendler. *Combining k-Induction with Continuously-Refined Invariants*. Technical Report MIP-1503, Department of Computer Science and Mathematics, University of Passau, 2015.

[BHKW12] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. *Conditional Model Checking: A Technique to Pass Information between Verifiers*. Proc. FSE, pages 57:1–57:11. ACM, 2012.

[BHMR07] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. *Path Invariants*. Proc. PLDI, pages 300–309. ACM, 2007.

[BHT07] D. Beyer, T. A. Henzinger, and G. Théoduloz. *Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis*. Proc. CAV, LNCS 4590, pages 504–518. Springer, 2007.

[BHT08] D. Beyer, T. A. Henzinger, and G. Théoduloz. *Program Analysis with Dynamic Precision Adjustment*. Proc. ASE, pages 29–38. IEEE, 2008.

[BJKS15] M. Brain, S. Joshi, D. Kroening, and P. Schrammel. *Safety Verification and Refutation by k-Invariants and k-Induction*, pages 145–161. LNCS 9291. Springer, 2015.

[BK11] D. Beyer and M. E. Keremoglu. *CPAchecker: A Tool for Configurable Software Verification*. Proc. CAV, LNCS 6806, pages 184–190. Springer, 2011.

[BKW10] D. Beyer, M. E. Keremoglu, and P. Wendler. *Predicate Abstraction with Adjustable-Block Encoding*. Proc. FMCAD, pages 189–197. FMCAD, 2010.

[BL13] D. Beyer and S. Löwe. *Explicit-State Software Model Checking Based on CEGAR and Interpolation*. Proc. FASE, LNCS 7793, pages 146–162. Springer, 2013.

[BLN+13] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler. *Precision Reuse for Efficient Regression Verification*. Proc. ESEC/FSE, pages 389–399. ACM, 2013.

[BLW15] D. Beyer, S. Löwe, and P. Wendler. *Refinement Selection*. Proc. SPIN, LNCS 9232, pages 20–38. Springer, 2015.

[BNSV14] G. Brat, Jorge A. Navas, Nija Shi, and A. Venet. *IKOS: A Framework for Static Analysis Based on Abstract Interpretation*, pages 271–277. LNCS 8702. Springer, 2014.

[CGJ+03] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. *Counterexample-guided Abstraction Refinement for Symbolic Model Checking*. J. ACM, volume 50, pages 752–794. ACM, 2003.

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

[DDLM13] I. Dillig, T. Dillig, B. Li, and K. L. McMillan. *Inductive invariant generation via abductive inference*. Proc. OOPSLA, pages 443–456. ACM, 2013.

[EPG+07] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. *The Daikon System for Dynamic Detection of Likely Invariants*. Sci. Comput. Program., volume 69, number 1-3, pages 35–45. Elsevier, 2007.

[FG10] P. Feautrier and L. Gonnord. *Accelerated Invariant Generation for C Programs with Aspic and C2fsm.* Electron. Notes Theor. Comput. Sci., volume 267, number 2, pages 3–13. Elsevier, 2010.

[GKN15] A. Gurfinkel, T. Kahsai, and J. A. Navas. *SeaHorn: A Framework for Verifying C Programs (Competition Contribution)*, pages 447–450. LNCS 9035. Springer, 2015.

[GR09] A. Gupta and A. Rybalchenko. *InvGen: An Efficient Invariant Generator*. Proc. CAV, LNCS 5643, pages 634–640. Springer, 2009.

[HB12] K. Hoder and N. Bjørner. *Generalized Property Directed Reachability*. Proc. SAT, LNCS 7317, pages 157–171. Springer, 2012.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. *The synchronous data flow programming language LUSTRE*. P. IEEE, volume 79, number 9, pages 1305–1320. IEEE, 1991.

[Jan07] M. Janota. *Assertion-based Loop Invariant Generation*. 2007.

[JSS14] B. Jeannet, P. Schrammel, and S. Sankaranarayanan. *Abstract Acceleration of General Linear Loops*. Proc. POPL, POPL'14, pages 529–540. ACM, 2014.

[KM16] G. Karpenkov and D. Monniaux. *Formula Slicing: Inductive Invariants from Preconditions*. Proc. HVC. Springer, 2016.

[KMW16] E. G. Karpenkov, D. Monniaux, and P. Wendler. *Program Analysis with Local Policy Iteration*, pages 127–146. LNCS 9583. Springer, 2016.

[KT11] T. Kahsai and C. Tinelli. *PKind: A parallel k-induction based model checker*. Proc. PDMC, EPTCS 72, pages 55–62. EPTCS, 2011.

[LA04] C. Lattner and V. Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. Proc. CGO, pages 75–86. IEEE, 2004.

[LB04] S. K. Lahiri and R. E. Bryant. *Constructing Quantified Invariants via Predicate Abstraction.* Proc. VMCAI, LNCS 2937, pages 267–281. Springer, 2004.

[LM10] K. R. M. Leino and R. Monahan. *Dafny Meets the Verification Benchmarks Challenge*, pages 112–126. LNCS 6217. Springer, 2010.

[MWK+15] K. Madhukar, B. Wachter, D. Kroening, M. Lewis, and M. K. Srivas. *Accelerating Invariant Generation*. Proc. FMCAD, pages 105–111. IEEE, 2015.

EIDESSTATTLICHE ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit 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 Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, den 9. Oktober 2016

Thomas Stieglmaier