Ludwig Maximilian University of Munich

Bachelor Thesis
in Computer Science

Correctness Witness Validation using Predicate Analysis

Maximilian Wiesholler

Thesis submitted to the Software and Computational Systems Lab

Supervisor: Prof. Dr. Dirk Beyer

Advisor: Martin Spießl

June 5, 2019

Statement of Originality

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. This paper was not previously presented to another examination board and has not been published.

Ottobrunn, June 5th 2019

.....
Maximilian Wiesholler

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Ottobrunn, 5. Juni 2019

.....
Maximilian Wiesholler

Abstract

Witness validation is an important step for increasing the confidence in the results of software verification. This is especially true when the verifier claims that the specification holds, i.e., when it outputs a correctness witness. While there are currently two validators for correctness witnesses, it is generally desirable to add new, conceptually different algorithms for validation. The goal of this thesis is to explore how predicate abstraction can be used in that regard. Possible approaches include reusing the invariants in the witness for the initial predicate precision or adding them as additional verification goals to the original verification task.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software is a fundamental component in today's society. As new requirements are demanded by industry and costumers, the complexity of software continuously increases and challenges the implementation of robust programs. Software has to fulfill the significant criterion of safety e.g. in vulnerable systems like autonomous driving or pacemakers. Incorrect behavior of software can lead to bad consequences for humans.
Testing is a common technique to increase the confidence in a program. By applying a wide range of test cases to the program, errors are detected and can be repaired. However, testing can not guarantee that the program is correct and minimizes only the likelihood of errors. Formal software verification is an alternative approach which tries to prove the (in-)correctness of a program. In order to verify a program a specification is first chosen which the program must fulfill. A specification describes properties in the program, for instance memory release after termination or the unreachability of a program line. If the property holds the program is correct regarding the specification, if the property does not hold the program is incorrect regarding the specification.
A verification tool (verifier) for formal verification can additionally output a witness. The type of the witness depends on the verification result. If the verifier claims that the program is correct it can produce a correctness witness which is a partial proof and can contain important invariants which have been found during the analysis. If the verifier claims that the program is incorrect it can produce a violation witness which summarizes paths to the property violation.
Witnesses can be validated. A verification tool with the ability to validate (validator) tries to reestablish the result. It receives the witness as input and analyzes the contained information. Since witnesses have a standardized exchange format, they can be validated independently from the tool which generated them. This enables sharing and validating of witnesses among tools and increases the confidence in the result when different validators return the same result.
The objective of this thesis is to use predicate analysis as new approach for validating correctness witnesses. The implementation is done in the

verification framework CPAchecker. Predicate analysis is based on the concept of predicate abstraction and already used in CPAchecker for verification. Regarding the validation of correctness witnesses, CPAchecker uses k-induction as validation method so far [2]. By using predicate analysis as an alternative validation method we will compare both methods. Furthermore, we will compare correctness witness validation of predicate analysis in CPAchecker with the verification tool Ultimate Automizer which can also validate correctness witnesses [2].

Related Work. The concept of producing correctness witnesses and validating them among different verification tools is relatively new. Concepts have been introduced in [2] which continue from work that has been done for the validation of violation witnesses [4]. Furthermore, approaches to validate correctness witnesses in CPAchecker and Ultimate Automizer have been presented in [2]. We extend the validation of correctness witnesses by using predicate analysis as technique. Since our work is done in CPAchecker, we can reuse implementations of reading and producing correctness witnesses in CPAchecker [2].
For validating correctness witnesses by using predicate analysis we present two approaches.
In the first approach we extract the invariants from correctness witnesses and reuse them as precision information. Similar work has been made in [9]. In this work precision information is stored in a precision file after the verification of a program finishes and reused in a later verification run.
In the second approach we define the correctness witness invariants as additional verification goal. Therefore, we use the invariants to build a specification automaton so that predicate analysis can verify this automaton. This approach has similarities to the validation approach in Ultimate Automizer. In order to validate correctness witnesses Ultimate Automizer uses a modified control-flow automaton. This automaton is built by taking the original control-flow automaton of the program and extending it with the invariants so that the invariants can additionally be verified [2].

Chapter 2

Background

Before we present the implementation we give an overview about the theoretical background. First, we describe basic terms of software verification. After that we explain the functionality of predicate analysis. Subsequently, we inform about the structure and information of witnesses and show in particular the concept of correctness witnesses. Furthermore, we describe how predicate analysis produces correctness witnesses and describe the functionality of k-induction in CPAchecker and the automata based concept of Ultimate Automizer to produce and validate correctness witnesses. Finally, we explain precision reuse.

2.1 Specification and Reachability

Providing a specification means to define properties which are to be proved for a given program. For instance, for a program P with a variable $x \in X$ where X denotes the set of program variables, a possible specification is the safety property that x is always equal or greater than zero. The program meets the specification when no program location can be found where x is less than zero. Checking whether a program fulfills a given specification is called software model checking [14].
A program can have an (infinite) amount of concrete paths and concrete states. A concrete state represents the variable assignment at a certain program location. Each concrete path consists of a sequence of concrete states starting at the initial state of the program. A specification separates all concrete paths into two sets. One set contains those paths which fulfill the specification, the other set contains those which violate the specification. The paths in the latter set are called concrete error paths and the concrete states that violate the specification are called concrete error states. If the set of concrete error paths is empty the program is proven as correct regarding the given specification.
Finding an error state can be reduced to the reachability problem with the error state as target state. The reachability problem is asking the question whether a concrete target state in the program is reachable. A state is reachable if a feasible program path starting from the initial program

location exists that leads to the state.

2.2 Control Flow Automaton

A program can be translated into a control-flow automaton (CFA) [6]. The CFA is a directed graph $A = (L, l_{init}, G)$ with nodes as program locations and edges or transitions as relation between program locations. L is the set of of program locations and $l_{init} \in L$ the program entry location. G is the set of transitions. A transition $(l, ops, l') \in G$ describes the operation ops that is executed if the control flow goes from location l to target location l'. The operation can be either an assignment or an assumption because we restrict the presentation to a simple imperative programming language [6]. Let x be a program variable of the set X of all program variables. An assignment has the form $x := e$ for which the value of the arithmetic expression e is assigned to x. An assumption $[\rho]$ with the predicate ρ over variables of X is evaluated at control statements and loop heads to guide the control flow.

Fig. 2.1 shows a program[1] taken from SV-COMP 2019 and a corresponding CFA. The program has two variables `x` and `y` which are initialized with the same non-deterministic value and contains a while loop which increments `x` and `y` in sequence until `x` is greater than 1024. In line `9` an `ERROR` label is inserted. When we define as specification the unreachability of the `ERROR` label and a concrete path exists which reaches line `9` the program violates the specification. The program is safe since `x == y` always holds when the program reaches line `8`. It is obvious that `x != y` can never be true at line `8`. However, to show the correctness of the program a verification technique must effectively prove that `x == y` is a loop invariant which holds when the program enters the loop head. We will refer to this example several times in this work.

```
1  int main(void) {
2    unsigned int x =
         __VERIFIER_nondet_uint()
         ;
3    unsigned int y = x;
4    while (x < 1024) {
5      x++;
6      y++;
7    }
8    if(x != y){
9      ERROR: return 1;
10   }
11   return 0;
12 }
```

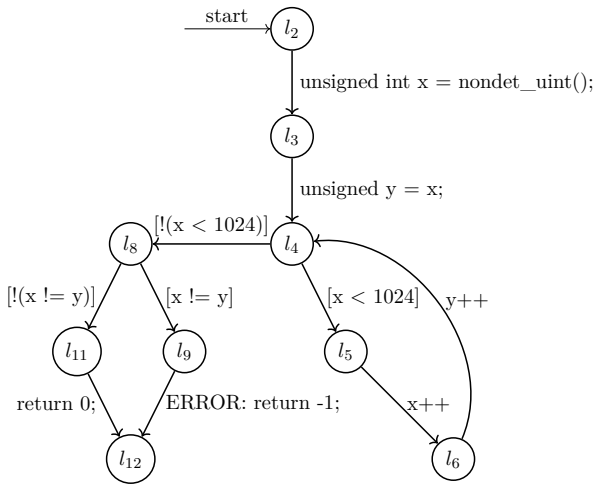

Figure 2.1: A program written in C and its corresponding CFA

1. https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/loop-acceleration/multivar_true-unreach-call1_true-termination.c

2.3 Abstract Model

We can create an abstract model of the program in form of an abstract reachability graph (ARG) [9]. For the abstract model we use abstract states that can overapproximate the reachable concrete program states. Each concrete state can be assigned to one or multiple abstract states and if an abstract state is reachable the concrete states covered by the abstract state are reachable as well. Since several concrete states might be represented by the same abstract state, abstraction can increase computation efficiency.
The abstract domain defines the abstract model. It specifies which program information the abstract states should contain.
The precision in an abstract domain describes the current stored information in an analysis. When we compute a successor state of an abstract state the computation is guided by the precision [9]. Often we try to get a balanced precision. A precision that is not accurate enough can lead to false alarms during the verification. A precision that is too accurate, on the other hand, might increase computation costs unnecessarily [9].
The ARG represents the abstract model. It is a directed graph and consists of reachable abstract states and transitions. The ARG is built iteratively by going through the CFA and applying the operations of the CFA on abstract states to compute abstract successor states.

2.4 Configurable Program Analysis

We can use a configurable program analysis (CPA) to define the abstract domain. A configurable program analysis algorithm (CPA algorithm) can perform a reachability analysis for a given CPA [6].

A CPA is a tuple $(D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ [6].

Abstract Domain. $D = (\mathcal{C}, \mathcal{E}, [[\cdot]])$ is the abstract domain with a set $\mathcal{C}$ of concrete states, a semi-lattice $\mathcal{E} = (E, \sqsubseteq, \sqcup, \top)$ with the set of abstract states E and the concretization function $[[\cdot]]$. The semi-lattice $\mathcal{E}$ defines the relation of abstract states with the partial order $\sqsubseteq$ between two states, the join $\sqcup$ of two states and the join over all states $\top$. The concretization function $[[\cdot]]$ computes for an abstract state $e \in E$ all concrete states it represents.

Precision. Π is the set of precisions used for the analysis. $\pi \in \Pi$ describes the precision used for an abstract state.

Transfer Relation. $\rightsquigarrow \subseteq E \times G \times E \times \Pi$ defines the transfer relation between two states of E with respect to the set of CFA edges G and the set of precisions Π. A transfer between two states e and e' exists if $(e, g, e', \pi) \in \rightsquigarrow$ and $g \in G$ and $\pi \in \Pi$. This means that e' is a reachable state from e taking the control-flow edge g using the precision π.

Operations. $\mathsf{merge} : E \times E \times \Pi \to E$ merges two states. If defined and possible, merge merges $e \in E$ and $e' \in E$ to a state $e_{new} \in E$.
$\mathsf{stop} : E \times 2^E \times \Pi \to \mathbb{B}$ checks if an abstract state $e \in E$ is covered by a set of reached abstract states 2^E under the precision $\pi \in \Pi$. If there exists a subset of abstract states which covers e stop returns true, otherwise it returns false.
$\mathsf{prec} : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$ recomputes the precision. For a state $e \in E$ with precision $\pi \in \Pi$ and a given set of reached abstract states prec adjusts the precision for e.

2.4.1 Configurable Program Analysis Algorithm

Algorithm 1 CPA++($\mathbb{D}$, reached, waitlist, abort)

Input: a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, merge, stop, prec)$ where Π denotes the set of precisions and where E denotes the set of elements of the semi-lattice of D
a set reached $\subseteq E \times \Pi$ of reachable abstract states
a set waitlist $\subseteq E \times \Pi$ of reached abstract states which are not yet processed
a function abort : $E \to \mathbb{B}$ that possibly aborts the algorithm
Output: the updated set reached and the updated set waitlist
Note: the technique of forced covering is omitted here

```
1:  //Main loop
2:  while waitlist ≠ ∅ do
3:      pop (e, π) from waitlist;
4:      for each e' with ê ⇝ (e', π) do
5:          (ê, π̂) := prec(e', π, reached);
6:          for each (e'', π'') ∈ reached do
7:              e_new := merge(ê, e'', π̂);
8:              if e_new ≠ e'' then
9:                  waitlist := (waitlist ∪ {(e_new, π̂)}) \ {(e'', π'')};
10:                 reached := (reached ∪ {(e_new, π̂)}) \ {(e'', π'')};
11:         if ¬stop(ê, {e|(e, ·) ∈ reached}, π̂) then
12:             waitlist := waitlist ∪ {(ê, π̂)};
13:             reached := reached ∪ {(ê, π̂)};
14:             if abort(ê) then
15:                 return (reached, waitlist);
16: return (reached, waitlist);
```

The CPA algorithm, more precisely the CPA++ algorithm[2] conducts a reachability analysis on a CPA [6].
The set waitlist stores abstract states and their corresponding precisions which have been reached but not processed so far. The set reached stores all reachable abstract states and their corresponding precision that have been found so far. The CPA++ algorithm is wrapped into another algorithm that conducts counterexample-guided abstraction refinement (CEGAR) and calls the CPA++ algorithm. CEGAR checks the satisfiability of reached when CPA++ returns reached. CEGAR then might refine the precision and call the CPA++ algorithm again. We explain CEGAR more detailed in 2.7.
In each loop iteration a pair (e, π) is picked from waitlist. In a for loop each successor e' is computed based on the transfer relation $\rightsquigarrow$. First, the precision of the abstract state is adjusted using the prec operator. In the nested second for loop each new successor is merged with states e''

2. The CPA++ algorithm is used for predicate analysis as an extension of the CPA algorithm. Both algorithms perform a reachability analysis

from reached. If a merge can be applied depends on the implementation of merge and whether merge conditions of the abstract states are satisfied. If $e_{new} \neq e''$ holds because merging has lead to a merged state e_{new}, $(e_{new}, \widehat{\pi})$ is inserted into reached and waitlist whereas (e'', π'') is removed. For each successor $\widehat{e}$ with $\widehat{\pi}$ it is checked whether another state from reached covers $\widehat{e}$. If not, $\widehat{e}$ is a new abstract state that is inserted into reached and waitlist. abort takes as argument $\widehat{e}$ and checks whether $\widehat{e}$ is an abstract error state. If that is the case the algorithm returns the current waitlist and reached. The algorithm returns the set of all reachable abstract states when the waitlist is empty and no abstract error state has been found before.

2.5 Predicate CPA

Predicate abstraction is an abstract interpretation technique which is used in software verification. It computes predicates over program variables in order to abstract concrete states and their assigned variables. The predicates are expressed as first-order formulas [9] which can be solved by a SMT solver.

Adjustable Block Encoding (ABE). ABE is a concept to compute abstractions only at locations where it is necessary [8]. Let ψ denote the abstraction formula of an abstract state and let φ denote the path formula which is a conjunction of several program operations. Instead of computing ψ for each program location, ψ is only computed at locations where necessary, for example at loop heads or control statements. When we compute ψ' as next abstraction, we call the SMT solver to combine the information from ψ and φ. For instance, for a sequence of assignments we do need to compute abstraction for each assignment. Instead, the sequence can be grouped into a block by conjugating the assignment operations in φ and will be evaluated together with ψ at the end of the sequence. Consequently, we need less solver calls without having information loss.

Predicate CPA. The *Predicate CPA* provides a predicate-based abstract domain for a reachability analysis.

The *Predicate CPA* $\mathbb{P}$ is a tuple $(D_{\mathbb{P}}, \Pi_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \mathsf{merge}_{\mathbb{P}}, \mathsf{stop}_{\mathbb{P}}, \mathsf{prec}_{\mathbb{P}})$ [6].

Abstract States. An abstract state is a triple $(\psi, l^{\psi}, \varphi)$ where ψ denotes the abstraction, l^{ψ} denotes the location of the abstraction formula and φ denotes the path formula.
Since *Predicate CPA* is based on the concept of ABE, it separates abstract states into *abstraction states* and *intermediate states*. *Abstraction states* satisfy always $\varphi = true$ because the abstraction has been computed. Intermediate states which are created between abstract states are grouped into a block. Their abstraction formulas refer to the last computed *abstraction state*. For two intermediate states $(\psi_1, l_1^{\psi}, \varphi_1)$ and $(\psi_2, l_2^{\psi}, \varphi_2)$ in the same

block the following holds:

$\psi_1 = \psi_2$ The abstraction formula is the same
$l_1^\psi = l_2^\psi$ The location of their abstraction formula is the same

Abstract Domain. $D_\mathbb{P} = (\mathcal{C}, \mathcal{E}_\mathbb{P}, [[\cdot]]_\mathbb{P})$ is the abstract domain with a set $\mathcal{C}$ of concrete states, a semi-lattice $\mathcal{E}_\mathbb{P}$ over the abstract states E and the concretization function $[[\cdot]]_\mathbb{P}$. An abstract state e of E represents a set of concrete states using the concretization function: $[[(\phi_e, l_e^\phi, \varphi_e]]_\mathbb{P} := \{(c, \cdot) \in C \mid c \models (\phi_e \wedge \varphi_e)\}$. For two abstract states e_1 and e_2, e_2 is stronger when $(\psi_{e1}, l_{e1}^\psi, \varphi_{e1}) \sqsubseteq (\psi_{e2}, l_{e2}^\psi, \varphi_{e2}) = ((\psi_{e1} \wedge \varphi_{e1}) \Rightarrow (\psi_{e2} \wedge \varphi_{e2}))$ holds.

Precision. A precision $\pi \in \Pi_\mathbb{P}$ is a mapping from program locations to sets of predicates over the program variables [6]. $\Pi_\mathbb{P}$ is the set of possible predicate precisions. $\pi(l)$ is the predicate precision at a program location l.

Transfer Relation. The transfer relation takes an edge of the CFA and computes always an *intermediate state* of an abstract state. It only changes the path formula and no abstraction computations are done, so in general: $(\psi, l^\psi, \varphi) \rightsquigarrow_\mathbb{P} ((\psi, l^\psi, \varphi'), \pi)$.

Merge. The $\mathsf{merge}_\mathbb{P}$ operation can only be applied to *intermediate states* in the same block:

$$\mathsf{merge}_\mathbb{P}((\psi_1, l_1^\psi, \varphi_1), (\psi_2, l_2^\psi, \varphi_2), \pi) =$$
$$\begin{cases} (\psi_2, l_2^\psi, \varphi_1 \vee \varphi_2) & \text{if } (\psi_1 = \psi_2) \wedge (l_1^\psi = l_2^\psi) \\ (\psi_2, l_2^\psi, \varphi_2) & \text{otherwise} \end{cases}$$

Stop. The $\mathsf{stop}_\mathbb{P}$ operator checks coverage only for *abstraction states*. An *abstraction state* is covered by another *abstraction state* from a set of abstract states when it is weaker or equal:

$$\mathsf{stop}_\mathbb{P}((\psi, l^\psi, \varphi, R, \pi) =$$
$$\begin{cases} ((\psi', l^{\psi'}, \varphi') \in R : \varphi' = true \wedge (\psi, l^\psi, \varphi) \sqsubseteq_\mathbb{P} (\psi', l^{\psi'}, \varphi') & \text{if } \varphi = true \\ false & \text{otherwise} \end{cases}$$

Prec Operator. The $\mathsf{prec}_\mathbb{P}$ uses boolean predicate abstraction $(\phi)_\mathbb{B}^\rho$ where ϕ denotes a formula and ρ a set of predicates. It applies predicate abstraction on *intermediate states* to get *abstraction states*: It takes the last abstraction formula ψ and the path formula φ as conjunction and computes the strongest boolean combination of predicates from the precision. The path formula gets *true*. blk defines at which locations *abstraction states*

should be computed. Its concrete configuration can be chosen. For instance, when selecting blk^l, *abstraction states* are computed at loop heads and at the error location.

$$\mathsf{prec}_{\mathbb{P}}((\psi, l^{\psi}, \varphi), \pi, R) = \begin{cases} (((\psi \wedge \varphi)^{\pi(l)}_{\mathbb{B}}, l, true), \pi) & \text{if } \mathsf{blk}((\psi, l^{\psi}, \varphi), l) \\ ((\psi, l^{\psi}, \varphi), \pi) & otherwise \end{cases}$$

Refinement. For the refinement process $\mathsf{refine}_{\mathbb{P}}$ takes the abstract states from reached [6]. First, it builds an abstract counterexample using the *abstraction states* in reached that lead to the abstraction error state. After that it constructs a sequence of abstraction formulas as conjunction from the *abstraction states* of the counterexample. The resulting counterexample formula is then given as query to a SMT solver in order to check if the formula is satisfiable. $\mathsf{refine}_{\mathbb{P}}$ stops if the formula is satisfiable and the analysis has found a violation of the specification. If not, the abstract states are too coarse and a refinement is necessary. Therefore $\mathsf{refine}_{\mathbb{P}}$ continues with interpolation in order to refine the abstract model. For the refinement $\mathsf{refine}_{\mathbb{P}}$ uses a given *refinement strategy* which modifies reached and waitlist. After that the reachability analysis starts again with the modified reached and waitlist.

Predicate Refinement. Predicate Refinement is a *refinement strategy* that refines the precision only at locations where it is necessary [6] (see Lazy Abstraction in 2.7). From the interpolation we obtain a sequence of sets of predicates that can be mapped to the program locations. At each location the current set of predicates is joined with the new set of predicates. After constructing the new precision we look at the first abstract state (pivot state) in the abstract counterexample for which the set of predicates has been updated. The pivot state and all its descendants are removed from the ARG so that the reachability analysis does not need to explore the whole state space again [6].

2.6 Composite CPA

Several CPAs can be combined to components by using a *Composite CPA* [6]. A state of the *Composite CPA* is a tuple that wraps one state of each component CPA. The precision of a *Composite CPA* state is a tuple which wraps the precisions of the component CPA states.
When the ARG is built the operations of the CFA are applied to abstract states of the *Composite CPA*. In order to compute the transfer relation the predecessor abstract state of the *Composite CPA* delegates the operation to each of its component states. The component states compute their successors respectively. By building the cartesian product of the successors the successor for the composite state is obtained.

By using a *Composite CPA* we can use CPAs that can focus on a specific task and can combine the information they track. Furthermore, such a composite pattern gives flexibility for different verification approaches.

2.6.1 Important Component CPAs

Location CPA. Since many analyses need information about the program counter, the *Composite CPA* often includes the *Location CPA* $\mathbb{L}$ [6]. It explicitly tracks the program counter. For instance, the *Composite CPA* contains a *Predicate CPA* and a *Location CPA*. When the *Predicate CPA* detects an error state on a feasible path the *Location CPA* can be consulted to get the program location of the error.

Automaton CPA. The *Automaton CPA* enables a configurable program analysis for an *observer automaton* or a *control automaton*. An *observer automaton* and a *control automaton* are protocol automata formalized by $A = (Q, \Sigma, \delta, q_0, F)$ [3]. Q is the set of states and q_0 is the initial state. Σ defines the alphabet over the relations between states. $\delta \subseteq (Q \times \Sigma \times Q)$ is the set of transfer relations. F is the set of accepting states.
An *observer automaton* monitors the state space exploration but does not affect it [5]. For instance, we can use an *observer automaton* to keep track of the specification by monitoring the reachability of an `ERROR` label in a program. A *control automaton* affects the state space exploration [5]. It can contain assumptions to guide and restrict the computation of abstract states.
The *Automaton CPA* for an *observer automaton* or *control automaton* can be used as a component of a *Composite CPA*.

2.7 CEGAR with Predicate Abstraction

Counterexample-guided Abstraction Refinement (CEGAR). CEGAR [10] is a technique which enables model checking to find iteratively a suitable precision [9]. In each iteration it invokes a nested reachability analysis that uses the current precision to build an abstract model.
An iteration in CEGAR works as follows [6]:
CEGAR calls the reachability analysis to construct the abstract model. The analysis uses the current precision. If an abstract error state is found during the construction the concrete path to this state is built in order to analyze if this counterexample is feasible. If the counterexample is feasible, the specification is violated. CEGAR terminates and the program does not satisfy the specification. If the counterexample is not feasible, the precision has been too coarse. With the information from the feasibility check the precision is refined so that the counterexample can be excluded in the next time. After the refinement CEGAR starts the next iteration.

If no abstract error state has been detected during the whole construction, the ARG is proven as safe. The program satisfies the specification.
By using CEGAR we try to find a suitable precision which is neither too coarse and nor to accurate [9]. A precision that is too accurate tracks unimportant facts and increases computations. The abstract domain often starts with an empty precision that gets continuously refined during the iterations.

Combining CEGAR with Predicate Abstraction. Predicate abstraction can be combined with CEGAR. As component in a *Composite CPA* the *Predicate CPA* is applied on a CPA++ algorithm which executes a reachability analysis on it [6]. Further components of the *Composite CPA* are usually a *Location CPA* to track the program counter and a *Automaton CPA* for the specification. The CPA algorithm is inserted into CEGAR so that the precisions of the composite CPA components and hence the precision of the Predicate CPA are continuously refined by CEGAR. In the following the term predicate analysis describes this configuration which we extend in parts of our implementation.

Lazy Abstraction. Lazy abstraction [11] can increase the efficiency when using predicate abstraction with CEGAR [6]. When a spurious counterexample in the refinement process has been found new precision facts will only be stored at program locations where the precision facts are necessary to exclude the counterexample. Consequently, the precision can vary for the computation of abstract successor states. This can decrease computation effort [9]. Furthermore, the abstract model is not built from scratch again. Parts of the abstract model which do not need to be recomputed in order to rule out the spurious counterexample remain.

2.8 Correctness Witnesses

We can distinguish between a violation witness or a correctness witness.
The violation witness is a *control automaton* that restricts the state space [2]. The validator follows assumptions in the violation witness and validates the witness if it can replay the path to the error state.
The correctness witness is an *observer automaton* that does not restrict the state space of the program [2]. A state can be labeled with an invariant θ which is a boolean expression. When a witness validator analyses the witness and enters a state labeled with an invariant the validator must check whether the invariant indeed holds.

2.8.1 Format of Witnesses

On the syntactic level we store a witness in a GraphML file [2]. GraphML is an XML based format. An important characteristic of a witness file is its

uniform format which enables verifiers to share their witnesses among each other.
The file includes general information about:

- witness type (violation witness, correctness witness)
- producer
- language of the verified program (C)
- applied specification
- path to the verified program
- program hash to identify the program
- used computer architecture (32 bit, 64 bit)
- time of creation

Exchange Format of Correctness Witnesses. As exchangeable witness format we save the correctness witness as graph with nodes and edges [2]. A node represents a state of the witness automaton. An edge represents a transition of the witness automaton and has as attributes a source node and a target node. Furthermore, edges can be labeled with source-code guards. They directly refer to the program code and contain conditions that can be matched when the witness is analyzed by a validator. Compared to violation witnesses correctness witnesses do not have state-space guards since those restrict the state-space[2]. A node can be labeled with an invariant.

Source-Code Guards. For a correctness witness transition there exists several source-code guards[3] [2]:

`startline` - `endline`	The CFA edge must match with the program lines
`startoffset` - `endoffset`	The CFA edge must match with the program offsets
`control with condition-true`	The CFA edge must match with the head of a control statement and the statement is evaluated as true
`control with condition-false`	The CFA edge must match with the head of a control statement and the statement is evaluated as false
`enterFunction`	The CFA edge must enter another function
`returnFrom`	The CFA edge must return to the calling function
`enterLoopHead`	The CFA edge must enter the head of a loop statement

State-Space Guards. Transitions of a correctness witness are not allowed to have state-space guards [2]. For state-space guards in violation witnesses the guard `assumption` is used to restrict the state-space. This guard contains a boolean expression ρ. If a state s has a transition with an assumption $[\rho]$ that goes into a successor state s', the transition can only be taken when ρ is evaluated as *true*.

3. Violation witnesses uses the same source-code guards except `enterLoopHead`

2.8.2 Correctness Witness Automaton

On the semantic level we regard the correctness witness as *observer automaton* and call it correctness witness automaton. The automaton is a protocol automaton $A = (N, \Sigma, \delta, n_{init}, F)$ where N denotes the set of states with initial state n_{init}, Σ denotes an alphabet over source-code guards, $G \subseteq (N \times \Sigma \times N)$ denotes the set of transitions and F denotes the set of accepting states. A transition $(n, guards, n')$ is taken when the automaton stays in n and the *guards* (i.e. source-code guards) are satisfied. A state $n \in N$ can be labeled with an invariant θ which must be proved when a validator validates the witness.

Example of a Correctness Witness Automaton. In Fig. 2.2 an example of a correctness witness automaton for the program in Fig. 2.1 is shown. It is a simplified version of a witness which is created when the verifier can prove the task. `PL` represents the program line guards. For example, the transition (n_1 , `(PL=3, enterLoopHead)`, n_2) corresponds to the program operation that enters the head of the while loop. The witness state n_1 covers all program operations before entering the loop head and the witness automaton remains in this state by executing the self-transition o/w ("otherwise").

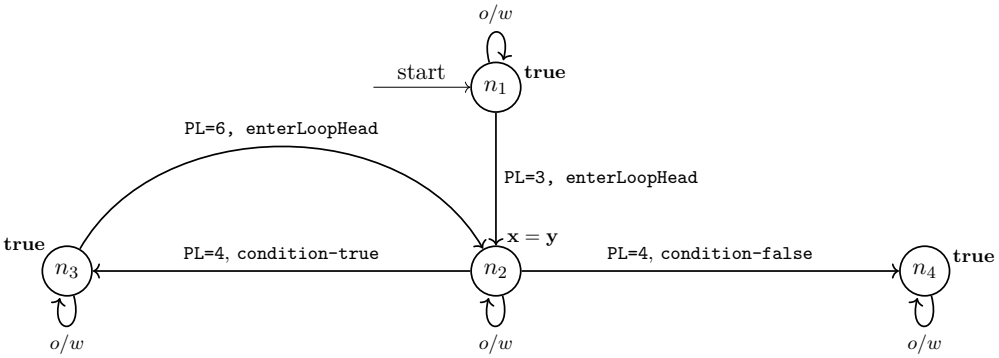

Figure 2.2: Example for a correctness witness automaton for the program from Fig. 2.1

Furthermore, we see that each state is labeled with an invariant. State n_2 is labeled with the *non-trivial* loop invariant $x = y$. This invariant has been found during the verification of the program. It must hold when the program proceeds to the head of the while loop. Then the automaton enters state n_2 and the invariant must be proved by the validator.
The transitions from n_2 to n_3 and n_4 respectively are taken based on the evaluation of the while condition of the program. The automaton takes the edge to n_3 if x is less than 1024 otherwise it takes the edge to n_4. The witness remains in n_3 (inside the while-loop) until a program operation proceeds to the loop head again. The automaton takes the transition to n_4 when x is not less than 1024 and will then remain in n_4 forever. The witness will take the self-transition o/w in n_4 for all program operations after the while loop in order to remain in n_4.

Note that we explicitly label each witness automaton state with an invariant. In general, a correctness witness on the syntactic level contains *non-trivial*-Invariants and does not need to label other states with the trivial invariant *true*. We say an invariant is *non-trivial* if it does not logically equals *true* and is expressed over variables of the program.

2.9 Approaches to Produce and Validate Correctness Witnesses

2.9.1 Producing Correctness Witness with Predicate Analysis in CPAchecker

Predicate analysis can produce a correctness witness when it labels the task as correct. In order to extract invariants it iterates over the edges of the CFA and gets for each target node the assigned abstract state. For each abstract state the predicate state is filtered and only predicate states are regarded further which are *abstraction states*. If the *abstraction state* has an abstraction formula that is also a valid boolean formula, the formula is translated into a C expression and written into the correctness witness. If a target of a CFA edge is assigned to more than one abstract state and each of the states return a C Expression we apply logic conjunction for the C expressions.

2.9.2 Producing and Validating Correctness Witnesses with *k*-induction in CPAchecker

k-induction in CPAchecker can produce correctness witnesses in a verification run. In addition, it has been extended to validate correctness witnesses [2].

Producing Correctness Witnesses. k-induction extends the technique of bounded model checking by combining it with induction to conduct unbounded safety checks [7].
Bounded model checking extracts all program paths that exists from a bounded unrolling of the program [6]. If a feasible path with an error state exists the specification is violated. However, bounded model checking can only find an error state for an unsafe program when the counterexample is detected within the given bound. It is limited when the program contains an unbounded loop like a while loop with a nondeterministic condition which executes a random amount of iterations.
With the concept of k-induction we can make unbounded model checking. For a specification S k-induction makes a bounded model check within bound k to check whether S holds and tries to proof S for consecutive steps in an induction phase [2].
Parallel to the analysis of k-induction an invariant generator is used which creates invariants and provides them as auxiliary invariants to the k-induction

analysis to strengthen the induction hypothesis [2]. The precision of the invariant generator increases as long as the analysis continues [7] so that the invariants get stronger. If k-induction can show the safety property, the auxiliary invariants can be written into a resulting correctness witness.

Validating Correctness Witnesses. CPAchecker with k-induction uses a preparatory step for the validation [2]. It matches the correctness witness with the CFA of the program so that each invariant of the witness is mapped to one or more CFA locations (see also section 6.1 in the appendix). The mapping to several program locations is based on the assumption that the witness might be imprecise [2]. After the preparation the analysis starts and CPAchecker applies the same k-induction algorithm that is used for the verification. Each k-induction iteration has two phases. In phase one the validator applies a bounded model check on the original safety property and the invariants. If the invariant can not be validated the invariant is discarded for the mapped location under the condition that there still exists other locations where the invariant might hold. If there is no other location mapped for the invariant the witness is violated. When the safety property can not be validated, the witness is violated as well. In phase two the validator applies k-induction on the safety property and the invariants. When it can prove the safety property the witness is accepted. Invariants which can be proved as inductive can be used as auxilarly invariants for following k-induction iterations [2].
It is to mention that for the validation analysis k-induction only uses a static invariant generator and can not create new invariants. The reason behind this design choice is not to validate witnesses that contain no *non-trivial*-Invariants [2]. In consequence, accepting a correctness witness with k-induction in CPAchecker strongly depends on the quality of the invariants in the correctness witness [2].

2.9.3 Producing and Validating Correctness Witnesses in Ultimate Automizer

Ultimate Automizer can produce correctness witnesses using automata-based verification. In addition, it has been extended to validate correctness witnesses [2].

Producing Correctness Witnesses. Ultimate Automizer constructs the CFA of the program and regards the CFA as automaton A_{CFA}. This automaton defines an alphabet over letters which represent the operations between CFA nodes [2]. A word is a path in the A_{CFA}. Accepted words are only those, which lead to an error state. Each accepted word must be infeasible for the correctness of the program. During the analysis Ultimate Automizer builds sequentially automata $A_1, ..., A_n$ which only accepts infeasible words. If the set of accepted infeasible words defined by the union of the automata $A_1, ..., A_n$ is a superset of the set of words accepted by

the A_{CFA} and each accepted automata word is infeasbile the program is proved as correct [2]. If, however, an automaton accepts a feasible word the program violates the specification.
If the program is proved as correct Ultimate Automizer creates invariants from the automata [2]. First, it computes the product of A_{CFA} and $A_1, ..., A_n$ to obtain a set of all reachable states. Each element of this set has the the form $(l, s_1, ..., s_n)$ where l denotes the CFA location and s denotes an automaton state. Each state s is labeled with a boolean formula which holds at s. In the second step Ultimate Automizer conjugates the formulas of elements which have the same location in order to receive a program invariant for each CFA location. Ultimate Automizer then only selects invariants which are loop invariants and writes them into the correctness witness.

Validating Correctness Witnesses. For the validation Ultimate Automizer uses two preparatory steps [2]. First Ultimate Automizer matches the correctness witness with the CFA of the program so that each invariant of the witness is assigned to its corresponding CFA location. After that it modifies the CFA using the invariants and their locations. Let l be a CFA location and let (l, op, l') be an outgoing edge of l. If l matches with one of the invariant locations Ultimate Automizer creates one transition to the new target locations $\widehat{l}$ and l_{err} respectively. l goes into $\widehat{l}$ if the invariant is assumed or goes into l_{err} if the negated invariant is assumed. For $\widehat{l}$ Ultimate Automizer creates an outgoing transition $(\widehat{l}, op, l')$ to l' the original target of l with op the original operation.
After the modified CFA is constructed Ultimate Automizer uses again automata-based verification by applying it on the modified CFA [2]. Ultimate Automizer accepts the witness when it can reestablish the proof of the original safety property and can confirm each invariant in the modified CFA.

2.10 Precision Reuse

Precision reuse describes the technique to reuse precision information for an abstraction-based analysis [9]. It is possible to store precision information in a program-precision file after the run of a program has finished. When the same program is analyzed again the precision can be initialized with the file content.
The motivation behind precision reuse is to decrease the necessary amount of refinements for the analysis and to decrease computation effort and time [9]. Each refinement needs computation costs because of feasibility checks and interpolation queries. Furthermore, when the precision is too coarse the ARG or parts of the ARG are constructed again. Using predicate abstraction the computation for abstract successors can be expensive since predicate abstraction sends queries to a SMT solver.

A precision can be distinguished regarding its scope: global-scoped precision, function-scoped precision and local-scoped precision [9]. When we reuse precision facts we add them to a certain scope. When adding them to the global-scope they are assigned to all program locations. When adding them to the function-scope they are assigned to all program locations of a certain function. When adding them to the local-scope they are only assigned to one certain program location.
Different reasons exist which precision scope should be chosen: Global-scoped and function-scoped precision are more robust against code changes [9]. For instance, after the first run the file stores a location-scoped precision that corresponds to the program line 6. Before the second run starts one new line of code has been inserted at line 6, hence the original line is shifted to line 7. In consequent, reusing the precision in a local-scope will not be helpful since it still corresponds to program line 6. Local-scoped precision, however, has the advantage that it avoids a mapping of precision facts to locations where such facts are not necessary. This might unnecessarily increase the computation effort. A precision initialization at a wrong location can be misleading and must be corrected during the refinement.

Chapter 3

Predicate Analysis based Correctness Witness Validation

We present two approaches to explore how predicate abstraction can be used to validate correctness witnesses. For our first approach we show a possibility to reuse the invariants from the correctness witness for the initial predicate precision. For our second approach we present three different concepts of a so-called invariant specification automaton (ISA). An ISA contains invariants from a correctness witness and gives us the opportunity to validate the invariants by using predicate analysis.

3.1 Location Invariants

A location invariant (l, θ) is a tuple where l denotes the CFA location and θ the invariant. I denotes the set of location invariants.
We need I for our approaches to initialize the predicate precision at CFA locations and to build two of our three proposed ISAs[1].
If we get a location invariant (l, θ) and it exists another location invariant $(l', \theta') \in I$ so that $l = l'$ holds we add the location invariant $(l, \theta \wedge \theta')$ to I and remove (l', θ') from I. This conjunction of two invariants is necessary to guarantee soundness.
In CPAchecker we can receive I by performing a preparatory step that includes several substeps: Parsing the correctness witness, performing a reachability analysis on the witness automaton, extracting location invariants from the reached set. We explain these steps in section 6.1 in the appendix.

1. The ISA^{WI} does not require location invariants for its construction

3.2 Predicate Precision Initialization with Witness Invariants

We can use the elements (l, θ) of the set I pf location invariants for the initial predicate precision. Each invariant θ is a C expression which we can transform into a predicate formula ρ that is understandable by a SMT solver.
Note that this validation approach only validates the original safety property from the correctness witness. It does not validate the invariants but only reuses them.

Precision Scope. The predicate precision can distinguish between local, function and global sets of predicates. Since we store for each element in I the CFA location l, we can add the invariants-based predicate to the local sets of predicates. In our evaluation, however, we will inspect all three possibilities and add the invariants-based predicates to the sets of local, function and global predicates respectively.
The following example shows the initialization of the predicate precision for each scope respectively for the program in Fig. 2.1 and its correctness witness in Fig. 2.2. From the witness we get $I = \{l_4, x = y\}$. Let L denote the set of CFA nodes. $\pi(l)$ maps a CFA location $l \in L$ to sets of predicates over the program variables. $\rho = (x = y)$ is the invariant-based predicate we get for location l_4.

- Adding the predicate to the location scope: $\pi(l_4) = \{\rho\}, \forall l \in L.l \neq l_4 : \pi(l) = \emptyset$
- Adding the predicate to the function scope: $\forall l \in L_f : \pi(l) = \{\rho\}$ where L_f denotes the set of locations in the scope of `main`.
- Adding the predicate to the global scope: $\forall l \in L : \pi(l) = \{\rho\}$

Using Atomic Predicates. We can additionally extract atomic predicates from a predicate. This technique splits the binary expressions of a predicate into components until each component is in atomic form. A predicate is in atomic form when a further split of this predicate would lead to components that are no boolean formulas anymore.
Let $\rho_1 = (a = 5 \wedge b = 2)$ denote a predicate. We can extract the atomic predicates $\rho_2 = (a = 5)$ and $\rho_3 = (b = 2)$. We can add these predicates to one of the scopes. For a CFA location $l \in L$ which is part of the scope the initial precision is then $\pi(l) = \{\rho_1, \rho_2, \rho_3\}$. Note that we also add ρ_1 since we might not be able to derive ρ_1 from ρ_2 and ρ_3 in the analysis.

Setup for Predicate Analysis with Invariants for Precision Reuse. We initialize the precision of the *Predicate CPA* with the invariants-based predicates and add it as a component to a *Composite CPA*. We apply the CPA++ algorithm with CEGAR to conduct an abstraction-analysis. We

will inspect whether the invariants are useful and whether the analysis can verify the orginal specification of the witness.

3.3 Witness Validation with an Invariants Specification Automaton (ISA)

In order to validate correctness witness we insert the witness invariants as additional verification goal into an ISA. Each ISA we present is a *control automaton* since it contains assumptions. In the following we show three different ISA concepts.

3.3.1 Two States Invariants Specification Automaton (ISA^{2S})

Let L denote the set of CFA locations. Let I denote the set of location invariants. Let Θ denote the set of invariants we get from the invariants in I. The ISA^{2S} is a protocol automaton $A = (S, \Sigma, \delta, s, F)$ with the set of automaton states S, the initial state $s := s_{init}$, the alphabet $\Sigma \subseteq (L \times \Theta)$, the transfer relation $\delta \subseteq (S \times \Sigma \times S)$ and the set of accepting states F. Θ denotes the set of invariants that contains the invariants from the elements in I, the invariants from the elements in I as negated invariants and the invariant *true*.
The automaton has two states: The initial state s and the error state E. E is an accepting state. For each location invariant (l, θ) we add two leaving transitions to s: $(s, l \wedge \neg[\theta], E)$ is the transition that goes into the error state assuming the negation of θ and $(s, l \wedge [\theta], s)$ is a self transition of s assuming θ. Furthermore, for each CFA node l' for which $l' \neq \widehat{\forall l} \in L_I$ holds the automaton stays in s using the self transition $(s, (l' \wedge [true]), s)$[2] respectively.

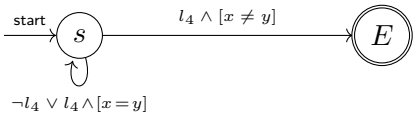

Figure 3.1: Example for an ISA^{2S} based on the program in Fig. 2.1 and the witness in Fig. 2.2

Fig. 3.1 illustrates an example of an ISA^{2S} based on the program in Fig. 2.1. The corresponding correctness witness in Fig. 2.2 has the invariant $x = y$ at witness node n_2. With the preparatory step we get $I = \{(l_4, x = y)\}$ which means that the invariant $x = y$ must hold when the CFA enters location l_4 ($\widehat{=}$ entering the while loop). Therefore, when the CFA provides a transition to l_4, the abstraction-analysis must assume that the invariant $x = y$ is valid otherwise it takes the transition $(s, (l_4 \wedge [x \neq y]), E)$ to the error state. For all CFA operations with a target location that does not equal l_4 the automaton stays in s.

2. In the concrete implementation we can omit this kind of transitions

3.3.2 CFA Based Invariants Specification Automaton (ISA^{CFA})

The ISA^{CFA} is another possibility of a specification automaton for invariants and has a structure that refers to the CFA. Let L be the set of CFA locations. Let I be the set of location invariants with $i = (\widehat{l}, \theta)$ where $\widehat{l}$ denotes the CFA location and θ the invariant. Let L_I be the set of locations of the invariants. Let Θ be the set of invariants we get from the invariants in I. The ISA^{CFA} is a protocol automaton $A = (S, \Sigma, \delta, s_{init}, F)$. S denotes the automaton states, s_{init} denotes the initial state, $\Sigma \subseteq (L \times \Theta)$ denotes the alphabet, $\delta \subseteq (S \times \Sigma \times S)$ denotes the automaton transitions and F denotes the accepting states. Θ denotes the set of invariants that contains the invariants from the elements in I, the invariants from the elements in I as negated invariants and the invariant *true*.

The automaton is constructed by using the CFA nodes and edges. A state $s \in S$ in the automaton refers either to one location node of the CFA or is an error state E or bottom state B. Each error state is an accepting state. s_{init} as initial state corresponds to l_{init} of the CFA. For each CFA edge of the set of CFA edges $G_{CFA} \subseteq (l \times ops \times l')$ we create a transition $(s, l' \wedge [true], s')$[3] if $l' \neq \forall \widehat{l} \in L_I$. For CFA edges for which $l' = \exists \widehat{l} \in L_I$ we create two transitions $(s, l' \wedge \neg[\theta], E)$ and $(s, l' \wedge [\theta], s')$. For a CFA node which has no successors the automaton goes into a bottom state: $(s, l' \wedge [true], B)$ if $succ(l') = \emptyset$. If the automaton enters a bottom state it will remain in this state forever. Note that also the special case $l' = \exists \widehat{l} \in L_I$ with $succ(l') = \emptyset$ might be possible. For this case we create two transitions: $(s, l' \wedge [\theta], B)$ and $(s, l' \wedge \neg[\theta], E)$.

Fig. 3.2 shows an example of a ISA^{CFA} based on the program in Fig. 2.1. We can see that for each CFA node we obtain a ISA^{CFA} state respectively. Since we get $I = \{(l_4, x = y)\}$, we construct for state s_2 and state s_5 two transitions respectively. For s_2 we create one transition that goes into the original successor s_3 assuming the invariant and one transition that goes into the error state assuming the negated invariant. For s_5 we create one transition that goes into the original successor s_3 assuming the invariant and one transition that goes into the error state assuming the negated invariant. The CFA location l_{12} has no successors, hence the ISA^{CFA} goes into a bottom state. Note the two leaving edges l_5 and l_8 of s_3: A CFA node might have more than one leaving edge. In consequence, the corresponding state of the ISA^{CFA} has for each leaving CFA edge a unique transition.

3. in the concrete implementation we can omit $[true]$

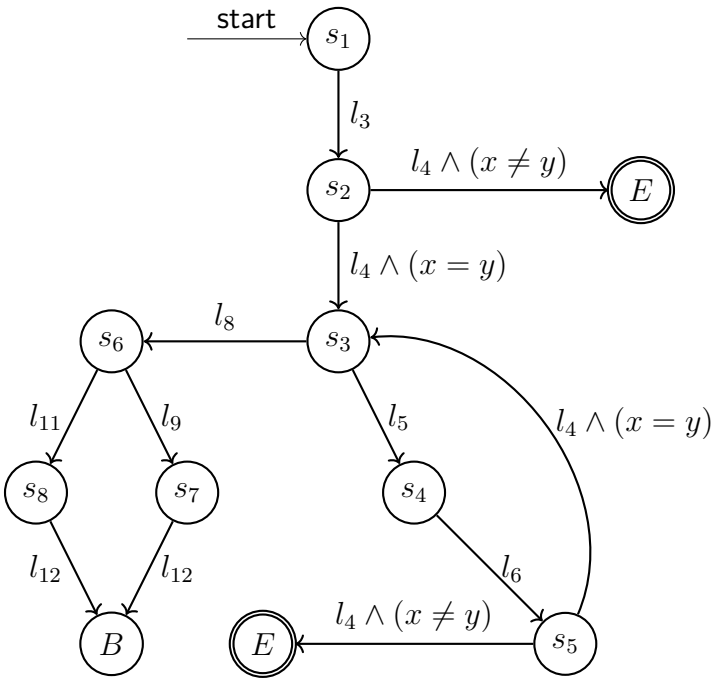

Figure 3.2: Example for an ISA^{CFA} based on the program in Fig. 2.1 and the correctness witness in Fig. 2.2

3.3.3 Witness Invariants Specification Automaton (ISA^{WI})

The ISA^{WI} is a protocol automaton $A = (S, \Sigma, \delta, s_{init}, F)$ with the set of states S, the initial state s_{init}, the alphabet $\Sigma \subseteq (Guards \times \Theta)$, the automaton transitions $\delta \subseteq (S \times \Sigma \times S)$ and the set of accepting states F. *Guards* are source-code guards of the witness. Θ is a set of invariants that contains the invariants from the witness, the invariants from the witness in negated form and the invariant *true*. Each error state E in ISA^{WI} is an accepting state.

Basically, the ISA^{WI} extends the witness automaton by using the invariants from Θ to build transitions that contain the invariants as (negated) assumptions. For each state in the witness we built a ISA^{WI} state s. For each transition in the witness we distinguish whether its target state s' is a state with a *non-trivial*-Invariant or not. If the invariant in s' equals `true` we built one ISA^{WI} transition $(s, (guards) \wedge [true], s')$[4]. If the invariant θ in s' does not equal `true` we built two ISA^{WI} transitions: $(s, (guards) \wedge [\theta], s')$ enters the original witness successor state assuming $[\theta]$ and $(s, (guards) \wedge \neg[\theta], E)$ goes into the error state E assuming $\neg[\theta]$. We add E to the set of states S. Note that each state of the correctness witness has a self transition. If such a state is labeled with a *non-trivial*-Invariant we built in the ISA^{WI} two self transitions using the invariant as assumption and negated assumption.

In particular, the ISA^{WI} does not need the preparatory step to receive the set of location invariants. We can apply the ISA^{WI} directly for the main predicate analysis.

Fig. 3.3 shows a ISA^{WI} based on program 2.1. We create two transitions when the original witness transition enters a state which is labeled with

4. In the concrete implementation we can omit [*true*]

an invariant. One transition goes into the original state assuming that the invariant holds. The other transition goes into an error state assuming the negated invariant. Since state s_2 is labeled with a non-trivial invariant, we create two self-transitions with assumptions that contain the (negated) invariant.

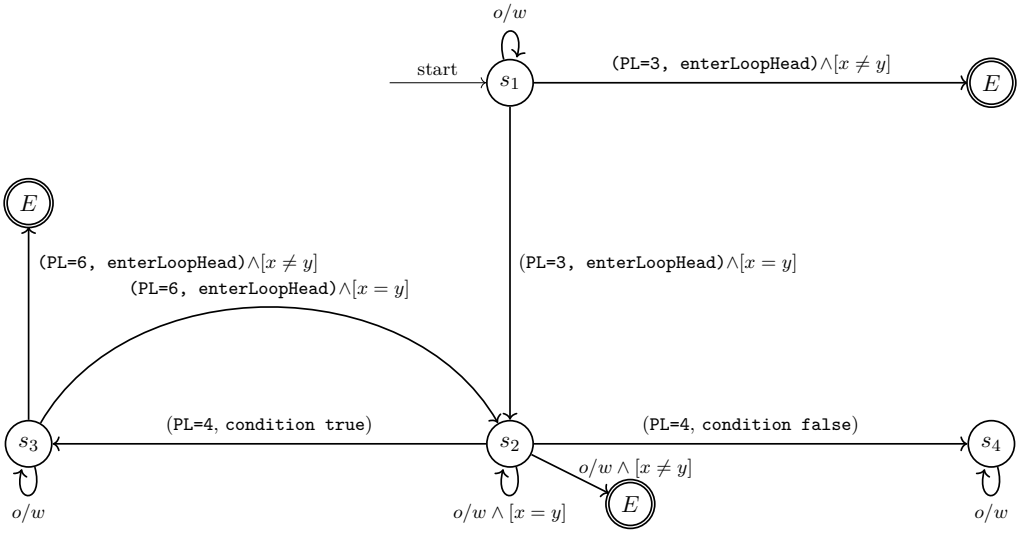

Figure 3.3: Example for an ISA^{WI} based on the program in Fig. 2.1 and the correctness witness in Fig. 2.2

3.3.4 Analysis with an ISA

Setup of Predicate Analysis with an ISA. Each of the presented ISA can be embedded in an *Automaton CPA*. The resulting *Automaton CPA* is added as component to a *Composite CPA*. The *Composite CPA* contains furthermore a *Predicate CPA* for predicate abstraction and a *Location CPA*. We use the CPA++ algorithm for an abstraction-analysis to build the ARG and insert the CPA++ algorithm into a CEGAR algorithm. This configuration allows us to use predicate analysis to validate the original safety property and the invariants of a correctness witness.

Refinement in Predicate Analysis. When an ISA stays in a state for which one of the successor states is an invariant based error state one of the two following cases will happen for an ISA during predicate analysis:
Case 1: The transition to the error state is taken because conditions are satisfied and the analysis assumes based on the current precision that the invariant is not valid. Hence, the CPA++ has found a target state and CEGAR checks the satisfiability of reached that contains the error state as last state. CEGAR will check whether the predicate precision is accurate enough. If the predicate precision is not accurate enough the precision is updated with the information from the refinement process and the (partial) construction of the ARG starts again. If the predicate precision is accurate enough the error state is reachable and we consequently have found an invariant which can not be verified. The analysis stops and the witness is

rejected.
Case 2: The transition to the error state is not taken because conditions are not satisfied or the current precision shows that the invariant holds. The CPA++ continues to process the waitlist.
Predicate analysis accepts the witness when the ARG has been completely built and has no reachable error states. This means that during the construction of the ARG no invalid invariant and also no violation of the safety property has been detected.
If a witness is rejected due to a reachable invariant-based error state, we say that the rejection is due to a witness invariant violation (WIV). The implementation in CPAchecker we use to detect a WIV is explained in section 6.2 in the appendix.

3.4 Types of Correctness Witnesses

Since invariants in correctness witnesses play an important role for our concepts, we distinguish three types of correctness witnesses:

non-trivial-witness	The correctness witness has states with *non-trivial*-Invariants and at least on of these states is reachable in the analysis.
true-witness	The correctness witness has no states labeled with *non-trivial*-Invariants.
hidden-true-witness	The correctness witness has states with *non-trivial*-Invariants but each of these states is unreachable in the analysis.

For a *true*-witness or a *hidden-true*-witness we can only verify the original safety property. The computation of location invariants I from a *true*-witness or *hidden-true*-witness leads to an empty set of I.
The types of correctness witnesses have the following consequences for our approaches:
For precision reuse of invariants a *true*-witness or *hidden-true*-witness leads to an empty set of predicates in the initial predicate precision π_0 since I is empty. A *non-trivial*-witness leads at least to one location for which the set of predicates is not empty.
The ISA^{2S} or ISA^{CFA} has no transitions into error states for a *true*-witness or *hidden-true*-witness since I is empty. The ISA^{WI} has no transitions into error states when it refers to a *true*-witness. However, if the ISA^{WI} refers to a *hidden-true*-witness, it has transitions into error states but these states are not reachable. The reason is that we directly take the unreachable states from the witness to create invariant-based error states in the ISA^{WI} without computing I. For a *non-trivial*-witness the ISA^{2S}, ISA^{CFA} and ISA^{WI} have transitions into reachable error states.
How we can detect *true*-witnesses and *hidden-true*-witnesses in our imple-

mentations in CPAchecker is explained in section 6.2 in the appendix.

Chapter 4

Evaluation

Procedure. In order to investigate how well predicate abstraction validates correctness witnesses we perform a large benchmark study.
Our evaluation is divided into three parts: The first part contains the necessary step to produce correctness witnesses. The second part covers the approach to reuse invariants from correctness witnesses for the initial predicate precision. In the third part we evaluate the ability of predicate analysis using an ISA to validate correctness witnesses and compare it with the k-induction-based validator in CPAchecker and the validator in Ultimate Automizer.

4.1 Abbrevations in the Evaluation

For better readability in the evaulation we use the following abbrevations:

pA	Predicate analysis in CPAchecker
kI	k-induction in CPAchecker
uA	Automaton-based analysis in Ultimate Automizer
x-Verification	Verification with analysis $x = (pA \mid kI \mid uA)$
x-`C`Witnesses	Correctness witnesses produced by analysis $x = (pA \mid kI \mid uA)$
x-`C`Witnesses$^{\text{no-}true}$	Correctness witnesses produced by analysis $x = (pA \mid kI \mid uA)$ which are no *true*-witnesses according to our implementation
x-`C`Witnesses$^{\text{no-}true^{+\mathtt{h}}}$	Correctness witnesses produced by analysis $x = (pA \mid kI \mid uA)$ which are no *true*-witnesses or *hidden-true*-witnesses according to our implementation
x-Invariants	Invariants of correctness witnesses produced by analysis $x = (pA \mid kI \mid uA)$
x-Validation	Validation with analysis $x = (pA \mid kI \mid uA)$
pA-Validation-`PR`	Validation with predicate analysis and reuse the witness invariants for predicate precision initialization
pA-Validation-`PR` $^{\mathtt{lo}\mid\mathtt{fu}\mid\mathtt{gl}}$	Adding the invariants-based predicates to the location(`lo`), function(`fu`) or global(`gl`) predicate precision scope

pA-Validation-PR *scope*+a	Adding the invariants-based predicates to the given scope additionally as atoms
pA-Validation-ISA^{2S}	Validation with predicate analysis using the ISA^{2S}
pA-Validation-ISA^{CFA}	Validation with predicate analysis using the ISA^{CFA}
pA-Validation-ISA^{WI}	Validation with predicate analysis using the ISA^{WI}
pA-Validation-ISA	Generic term to summarize *pA*-Validation-ISA^{2S}, *pA*-Validation-ISA^{CFA} and *pA*-Validation-ISA^{WI}

4.2 Results Explanation

In our evaluation we use the following status results by an analysis for a verification (marked with 1) or validation (marked with 2) task:

all	The sum of all tasks given for the analysis
correct-true[1]	Tasks which satisfy the specification and are labeled correctly by the analysis
correct-false[1]	Tasks which do not satisfy the specification but are labeled correctly by the analysis
incorrect-true[1]	Tasks which do not satisfy the specification and are labeled incorrectly by the analysis
incorrect-false[1]	Tasks which do satisfy the specification but are labeled incorrectly by the analysis
timeout[1,2]	Tasks for which the analysis exceeds the time limit of 900 s
error[1,2]	An error appears during the analysis leading to an unknown result
other[1,2]	Groups other reasons during the analysis that lead to an unknown result: memory errors, exceptions, verifier/validator explicitly returns unknown
accepted[2]	Tasks for which the analysis was able to validate the correctness witness
violated[2]	Tasks for which the analysis explicitly returns that the original specification or the invariants of the correctness witness is violated

In the following we only consider correctness witnesses for tasks for which the analysis that produces the correctness witnesses can label the tasks as correct-true.
We say that a correctness witness is accepted or confirmed when the analysis can validate the witness. On the contrary, we say that a correctness witness is rejected when the specification or invariants of the correctness witness are violated or when the analysis produces an error, a timeout or fails for another reason.
In the evaluation we sometimes compare a verification run with a validation run. Then we regard the result correct-true as equivalent to the validation result accepted.

4.3 Experiment Goals

Validation with Precision Reuse

Claim 1.1 *pA*-Validation-PR can validate the original safety property of *pA*-Witnesses and in particular does not violate *pA*-Witnesses.
Claim 1.2 Precision reuse of *pA*-Witnesses in *pA*-Validation-PR leads to less refinements and less CPU time since the predicate precision is initialized with useful predicates based on the invariants from *pA*-Witnesses.

Validation with Witness Invariants as Additional Specification
Claim 2.1 *pA*-Validation-$\texttt{ISA}^{2S}$, *pA*-Validation-$\texttt{ISA}^{CFA}$ and *pA*-Validation-$\texttt{ISA}^{WI}$ are able to validate the original safety property and the invariants of *pA*-Witnesses respectively.
Claim 2.2 *pA*-Validation-$\texttt{ISA}^{2S}$, *pA*-Validation-$\texttt{ISA}^{CFA}$ and *pA*-Validation-$\texttt{ISA}^{WI}$ are able to understand *kI*-Witnesses and *uA*-Witnesses.

4.4 Experimental Setup

Benchmarks. The set of benchmarks for our evaluation is taken from the category ReachSafety and category SoftwareSystems of SV-COMP 2019. We take tasks from the following subcategories:

- ReachSafety-Arrays
- ReachSafety-BitVectors
- ReachSafety-ControlFlow
- ReachSafety-ECA
- ReachSafety-Floats
- ReachSafety-Heap
- ReachSafety-Loops
- ReachSafety-ProductLines
- ReachSafety-Sequentialized
- Systems_DeviceDriversLinux64_ReachSafety

We exclude the subcategory ReachSafety-Recursive because our configuration of predicate analysis in CPAchecker can not handle recursion[1]. As specification we chose the unreachability of the error function __VERIFIER_error().
In general, the name of each tasks consists of the identifying name and the specification(s) it has to fulfill. All tasks from the subcategories satisfy the regular expression *_(false|true)-unreach-call* in their name. Since we focus on correctness witnesses and do not want to receive violation witnesses, we exclude all tasks that violate the specification, i.e. we exclude all tasks with names matching the regular expression *_false-unreach-call*. In total we have 4668 verification tasks for our evaluation.

1. Some tasks in Systems_DeviceDriversLinux64_ReachSafety and some few tasks in ReachSafety-Heap and ReachSafety-Loops contain also recursions so that predicate analysis will return recursion errors in the evaluation for these tasks.

Tools. The evaluation is made with BenchExec an open-source benchmarking framework [1]. It manages benchmarking to evaluate the performance of verification tools. It is independent from the verification tool and gives us the opportunity to produce and validate correctness witnesses under the same conditions for CPAchecker and Ultimate Automizer.
The CPAchecker version we use has revision number 31181. The version of Ultimate Automizer is taken from SV-COMP 2019[2]. For BenchExec we use the version integrated in CPAchecker with revision number 31181.

System Setup. Benchmarking is done on machines with 8 core CPUs with 3.40 GHz (Intel Xeon E3-1230 v5) and 33 GB of RAM memory. The operating system on the machines is Ubuntu 18.04 (64 Bit). For each verification task we set the following resource options: a processing time limit of 900 s, a memory limit of 15 GB and a requirement of 8 CPU cores.

Verification Setup. For *pA*-Verification and *kI*-Verification we switch off the options to aggregate basic blocks in the *Composite CPA* and to simplify the CFA. Both options can increase the efficiency but might affect the precision of witnesses. Apart from that we take the default settings CPAchecker provides for predicate analysis and *k*-induction. The verification configurations for *uA*-Verification are adopted from SV-COMP 2019[3].

Validation Setup. For all configurations of *pA*-Validation-`ISA` and for *kI*-Validation we switch off the option to check the program hash in the witness. For each *pA*-Validation-`ISA` we apply the same predicate abstraction settings which are used for *pA*-Verification. The specific settings we add in order to use different approaches of *pA*-Validation-`ISA` are summarized in Tab. 6.1 with a short explanation. For *kI*-Validation we adopt the default settings CPAchecker provides. For *uA*-Validation we adopt the validation configurations from SV-COMP 2019[4].

4.5 Generation of Correctness Witnesses

The verification results are presented in Tab. 4.2. We see that *pA*-Verification verifies 2396 tasks as correct-true and consequently we receive 2396 valid *pA*-Witnesses that we can use for validation. *kI*-Verification verifies 2599 tasks as correct-true and consequently we receive 2599 valid *kI*-Witnesses. *uA*-Verification is able to verify 2688 tasks as correct true and consequently we receive 2688 valid *uA*-Witnesses.
It is possible that the verification analyzes produce correctness witnesses also when they exceed the timeout. Such witnesses are always filtered when

2. `https://gitlab.com/sosy-lab/sv-comp/archives-2019/raw/svcomp19/2019/uautomizer.zip`
3. `https://github.com/sosy-lab/sv-comp/blob/svcomp19/benchmark-defs/uautomizer.xml`
4. `https://github.com/sosy-lab/sv-comp/blob/svcomp19/benchmark-defs/uautomizer-validate-correctness-witnesses.xml`

we evaluate validation runs.

Table 4.2: Verification results for *pA*-Verification, *kI*-Verification and *uA*-Verification.

status	all	correct-true	correct-false	incorrect-true	incorrect-false	timeout	error	other
pA-Verification	4668	2396	0	0	5	1533	696	38
kI-Verification	4668	2599	0	0	1	1784	134	150
uA-Verification	4668	2688	0	0	2	1684	90	204

Note that we also show the results correct-false and incorrect-true in Tab. 4.2. These results should not be returned since we only have tasks which are correct regarding the unreachability of the error function. However, we put correct-false and incorrect-true into the table for completeness.
The received correctness witnesses will be validated in different approaches in the following experiments.

4.6 Reusing Invariants from Correctness Witnesses as Initial Predicate Precision

In our first experiment for validation of correctness witnesses using predicate analysis we take the invariants from the witnesses and add them to the initial predicate precision (*pA*-Validation-PR). We observe whether *pA*-Validation-PR can reestablish the original safety property. Furthermore, we study how the precision-scope affects the computation and whether extracting the invariants-based predicates into atoms benefits the computation.
Our main focus in this experiment are invariants from *pA*-Witnesses. We have evaluated whether *uA*-Invariants or *kI*-Invariants are useful for the initial predicate precision. Fig. 4.1 illustrates the computation time between *pA*-Validation-PR$^{\text{lo}}$ and *pA*-Verification when we initialize the precision of *pA*-Validation-PR$^{\text{lo}}$ with *kI*-Invariants or *uA*-Invariants. Note that we only consider invariants from *kI*-Witnesses$^{\text{no-true}^{+\text{h}}}$ and *uA*-Witnesses$^{\text{no-true}^{+\text{h}}}$ since *true*-witnesses or *hidden-true*-witnesses would have no effect on the initial predicate precision in *pA*-Validation-PR$^{\text{lo}}$. Looking at Fig. 4.1 we assume that *kI*-Invariants are rather misleading as facts for the predicate precision and might be discarded or updated during the refinement. For *uA*-Invariants we can not say whether they might be helpful because the task set we can consider is extremely reduced. There are two reasons for this reduction: First, *pA*-Validation-PR$^{\text{lo}}$ detects many witnesses from *uA*-Verification that are *true*-witnesses or *hidden-true*-witnesses. Second, *pA*-Validation-PR$^{\text{lo}}$ and *pA*-Verification can sometimes not label tasks correct-true for which *uA*-Verification produces *kI*-Witnesses$^{\text{no-true}^{+\text{h}}}$.
Tab. 4.3 sums up our validation results using *pA*-Validation-PR for *pA*-Witnesses. We see that *pA*-Validation-PR can reestablish the result for nearly all witnesses. For two tasks each *pA*-Validation-PR can not parse

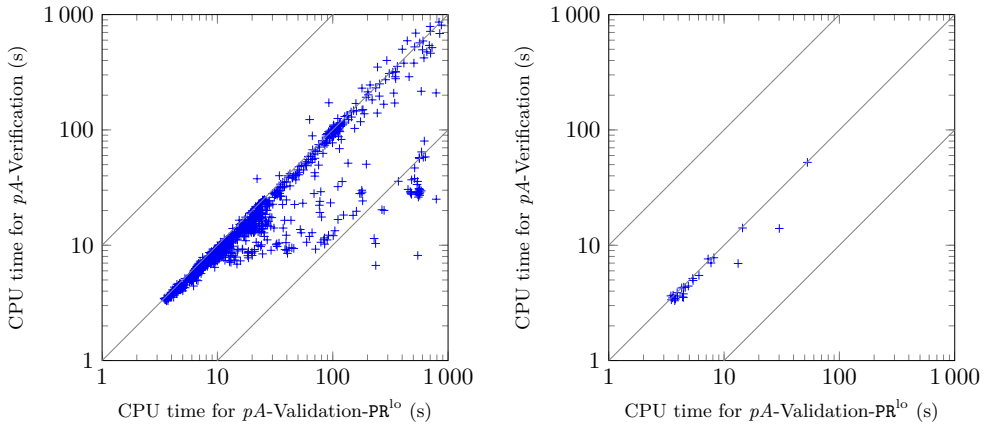

Figure 4.1: Scatter plots for comparing CPU time of pA-Validation-$\texttt{PR}^{\text{lo}}$ and pA-Verification for each task that both analyzes can label as correct-true. **Left Side:** pA-Validation-$\texttt{PR}^{\text{lo}}$ initialized with kI-Invariants from kI-Witnesses$^{\text{no-true}^{+\text{h}}}$. **Right Side:** pA-Validation-$\texttt{PR}^{\text{lo}}$ initialized with uA-Invariants from uA-Witnesses$^{\text{no-true}^{+\text{h}}}$.

the correctness witness because of an an exception[5] or interpreting the witness as invalid[6]. We can also see that using atomic predicates produces more often timeouts. The configuration for using atomic predicates is also applied for the main analysis and not only for the initialization of the predicate precision. The timeouts are mainly produced for tasks from subcategories `ReachSafety-ECA` and `Systems_DeviceDriversLinux64_-ReachSafety`. When we use atomic predicates we have in the analysis more predicates compared to an analysis that does not use atomic predicates. Having more predicates might increase computation costs.

Table 4.3: Status results when using pA-Invariants as predicates in the initial predicate precision

	pA-Validation-PR	pA-Validation-$\texttt{PR}^{\text{lo}}$	pA-Validation-$\texttt{PR}^{\text{lo+a}}$	pA-Validation-$\texttt{PR}^{\text{fu}}$	pA-Validation-$\texttt{PR}^{\text{fu+a}}$	pA-Validation-$\texttt{PR}^{\text{gl}}$	pA-Validation-$\texttt{PR}^{\text{gl+a}}$
2396 pA-Witnesses	accepted	2393	2351	2393	2352	2392	2349
	violated	0	0	0	0	0	0
	error	1	1	1	1	1	1
	timeout	1	43	1	42	2	45
	other	1	1	1	1	1	1
223 pA-Witnesses$^{\text{no-true}^{+\text{h}}}$	accepted	223	219	223	219	222	217
	violated	0	0	0	0	0	0
	error	0	0	0	0	0	0
	timeout	0	4	0	4	1	6
	other	0	0	0	0	0	0

Looking at pA-Witnesses$^{\text{no-true}^{+\text{h}}}$ in Tab. 4.3 we can say that pA-Validation-

5. `https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/ldv-linux-4.2-rc1/linux-4.2-rc1.tar.xz-43_2a-drivers--regulator--lp8755.ko-entry_point_true-unreach-call.cil.out.c`
6. `https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/bitvector/parity_true-unreach-call_true-no-overflow.c`

PR is able to reuse invariants-based predicates in the initial predicate precision[7]. Moreover, *pA*-Validation-PR never violates the original specification as expected.

We take a closer look at the computation time and amount of refinements between *pA*-Verification and *pA*-Validation-PR. For this part of the experiment we only regard *pA*-Witnesses$^{\text{no-true}^{+h}}$. Witnesses which do not have any invariants except *true* have no effect for our experiment since they lead to an empty initial predicate precision.

In Fig. 4.2 we see the number of CEGAR refinements and CPU time for *pA*-Validation-PR$^{\text{lo}}$ and *pA*-Validation-PR$^{\text{lo+a}}$ comparing it with *pA*-Verification respectively.

For *pA*-Validation-PR$^{\text{lo}}$ the invariants-based predicates have no effect on the number of refinements (see also Tab. 4.4). We also can observe a small tendency that the CPU time is higher for *pA*-Validation-PR$^{\text{lo}}$. This is probably caused by the preparatory step where we compute the set of location invariants by performing a reachability analysis.

For *pA*-Validation-PR$^{\text{lo+a}}$ there exists some few tasks for which the number of CEGAR refinements is reduced. However, fewer refinements does not decrease the CPU time. The atomic predicates increase the precision information and sometimes make the solver computations more costly. We can also observe again the small tendency that the CPU time is slightly higher for *pA*-Validation-PR$^{\text{lo+a}}$ because of the preparatory step.

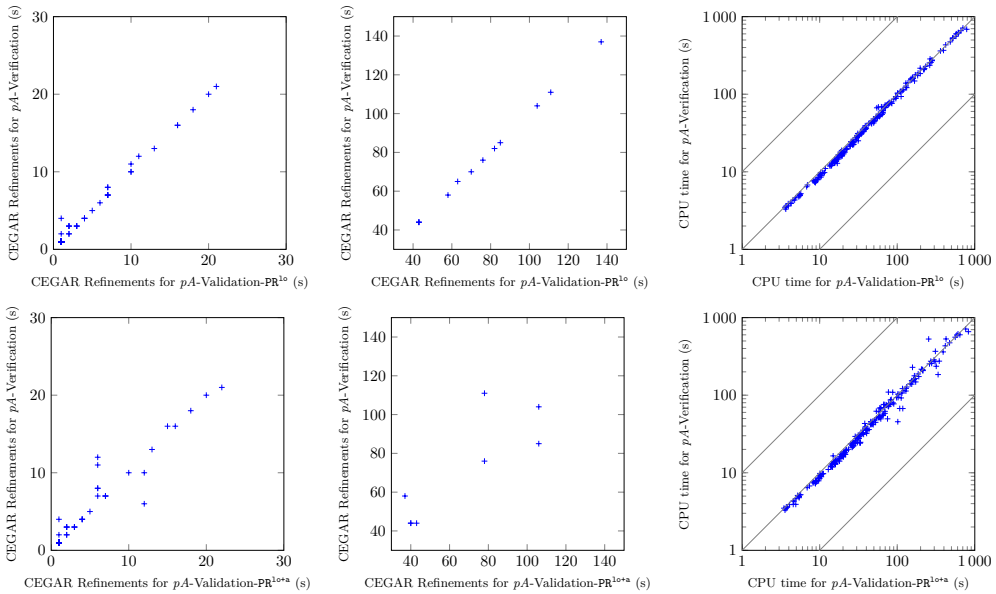

Figure 4.2: Scatter plots for comparing CEGAR refinements in interval $[0-30]$, CEGAR refinements in interval $[30-150]$ and CPU time. *pA*-Validation-PR$^{\text{lo}}$ and *pA*-Validation-PR$^{\text{lo+a}}$ precision initialized with *pA*-Invariants. **Upper row:** Comparing these values of *pA*-Validation-PR$^{\text{lo}}$ and *pA*-Verification for each task that both analyses labeled correct-true. **Lower row:** Comparing these values of *pA*-Validation-PR$^{\text{lo+a}}$ and *pA*-Verification for each task that both analyses labeled correct-true.

7. For 2 from 2396 *kI*-Witnesses we can not detect whether they contain *non-trivial*-Invariants because each *pA*-Validation-PR failed to produce statistics.

In Fig. 4.3 we see the number of CEGAR refinements and CPU time for *pA*-Validation-PR$^{\text{fu}}$ and *pA*-Validation-PR$^{\text{fu+a}}$ comparing it with *pA*-Verification respectively.
For *pA*-Validation-PR$^{\text{fu}}$ the invariants-based predicates reduce for few tasks the number of refinements. Again we see a tendency that the CPU time is slightly higher for *pA*-Validation-PR$^{\text{fu}}$ because of the preparatory step.
For *pA*-Validation-PR$^{\text{fu+a}}$ there exists some tasks for which the number of CEGAR refinements is reduced. The CPU time is often higher for *pA*-Validation-PR$^{\text{fu+a}}$ because of the greater amount of precision information.

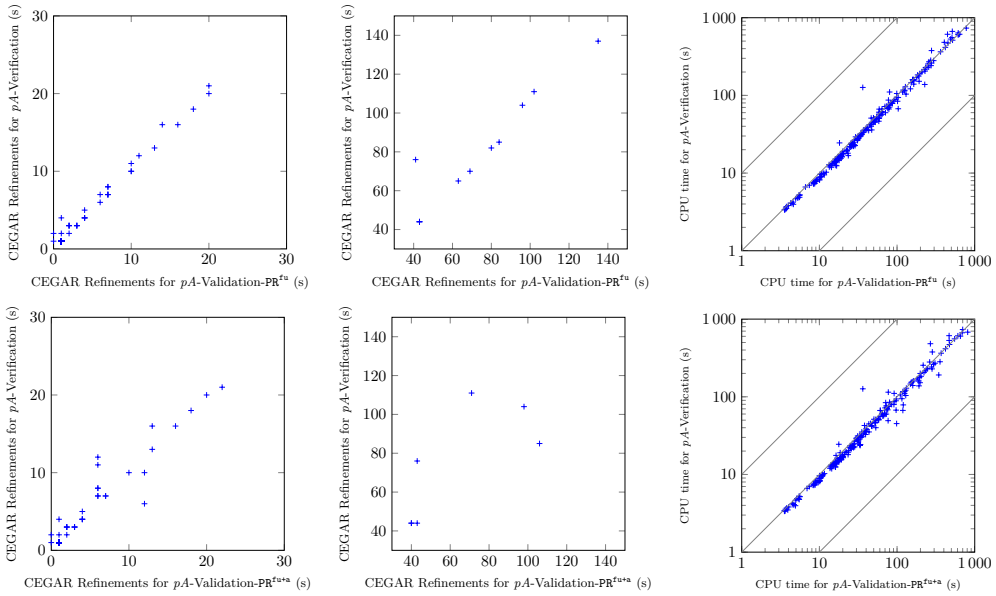

Figure 4.3: Scatter plots for comparing CEGAR refinements in interval $[0-30]$, CEGAR refinements in interval $[30-150]$ and CPU time. *pA*-Validation-PR$^{\text{fu}}$ and *pA*-Validation-PR$^{\text{fu+a}}$ precision initialized with *pA*-Invariants. **Upper row:** Comparing these values of *pA*-Validation-PR$^{\text{fu}}$ and *pA*-Verification for each task that both analyses labeled correct-true. **Lower row:** Comparing these values of *pA*-Validation-PR$^{\text{fu+a}}$ and *pA*-Verification for each task that both analyses labeled correct-true.

In Fig. 4.4 we see the number of CEGAR refinements and CPU time for *pA*-Validation-PR$^{\text{gl}}$ and *pA*-Validation-PR$^{\text{gl+a}}$ comparing it with *pA*-Verification respectively.
For *pA*-Validation-PR$^{\text{gl}}$ as well as *pA*-Validation-PR$^{\text{gl+a}}$ the invariants-based predicates reduce for several tasks the number of CEGAR refinements. Moreover, many tasks for *pA*-Validation-PR$^{\text{gl}}$ and *pA*-Validation-PR$^{\text{gl+a}}$ even need 0 refinements when we look at Tab. 4.4. Having the invariants-based predicates in the global predicate precision, predicate analysis tracks them for every location.
For the CPU time when comparing *pA*-Validation-PR$^{\text{gl}}$ and *pA*-Validation-PR$^{\text{gl+a}}$ with *pA*-Verification we can still see a line but see also tasks that are not located on the line. For the majority of tasks the initial precision information is not helpful and these tasks need slightly more CPU time because of the preparatory step. For a few tasks the initialized precision contains information that might be rather costly to track and increases

computation time. For some other tasks the invariants-based predicates might be indeed helpful and reduce computation time. Furthermore, it is possible that some of the tasks which are not located on the line can be due to random variances, caused e.g. by nondeterministic behavior during the validation.

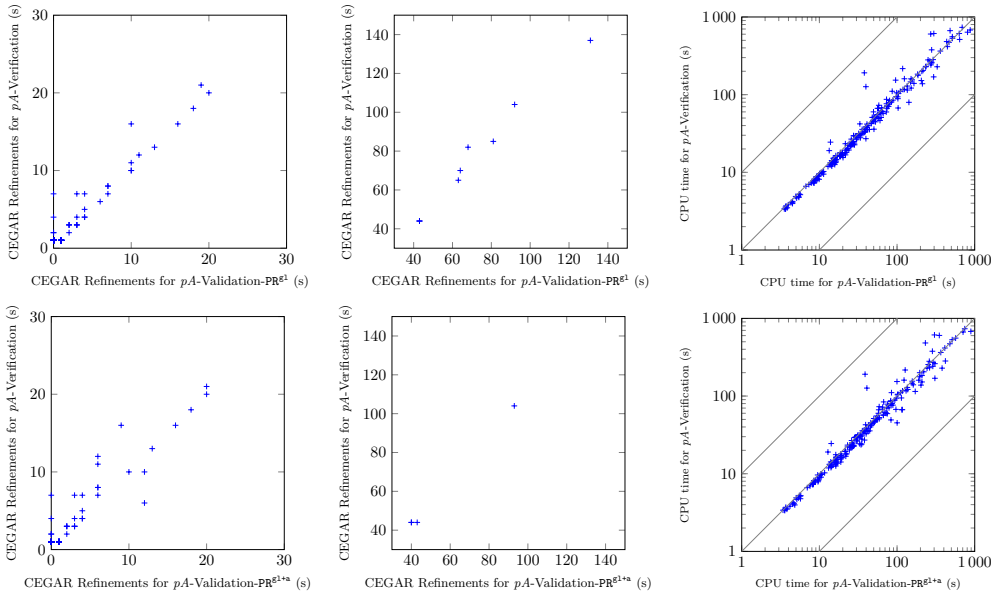

Figure 4.4: Scatter plots for comparing CEGAR refinements in interval $[0-30]$, CEGAR refinements in interval $[30-150]$ and CPU time. pA-Validation-$\texttt{PR}^{\text{gl}}$ and pA-Validation-$\texttt{PR}^{\text{gl+a}}$ precision initialized with pA-Invariants. **Upper row:** Comparing these values of pA-Validation-$\texttt{PR}^{\text{gl}}$ and pA-Verification for each task that both analyses labeled correct-true. **Lower row:** Comparing these values of pA-Validation-$\texttt{PR}^{\text{gl+a}}$ and pA-Verification for each task that both analyses labeled correct-true.

Table 4.4: Comparing number of CEGAR refinements for pA-Validation-PR approaches with pA-Verification for each task that both analyzes can label as correct-true. pA-Validation-PR approaches initialized with pA-Invariants.

CEGAR refinements	0	1	2	3	4	5	[6-10]	[11-20]	[21-30]	30<	Number of tasks
pA-Verification	0	180	3	6	4	1	9	7	1	12	223
pA-Validation-$\texttt{PR}^{\text{lo}}$	0	182	5	3	3	1	10	6	1	12	223
pA-Verification	0	179	3	6	4	1	9	7	1	9	219
pA-Validation-$\texttt{PR}^{\text{lo+a}}$	0	181	5	3	3	1	9	7	2	8	219
pA-Verification	0	180	3	6	4	1	9	7	1	12	223
pA-Validation-$\texttt{PR}^{\text{fu}}$	2	181	4	3	4	0	10	8	0	11	223
pA-Verification	0	179	3	6	4	1	9	7	1	9	219
pA-Validation-$\texttt{PR}^{\text{fu+a}}$	2	180	4	3	4	0	9	8	2	7	219
pA-Verification	0	180	3	6	4	1	9	7	1	11	222
pA-Validation-$\texttt{PR}^{\text{gl}}$	115	69	4	5	4	0	9	7	0	9	222
pA-Verification	0	179	3	6	4	1	9	7	1	7	217
pA-Validation-$\texttt{PR}^{\text{gl+a}}$	114	69	4	5	4	0	8	8	1	4	217

Summary. When we look at all different possibilities of pA-Validation-PR we observe that the CEGAR refinements are sometimes reduced when we initialize the predicate precision. However, we can not observe that this decreases the CPU time. For pA-Validation-$\texttt{PR}^{\text{lo}}$, pA-Validation-$\texttt{PR}^{\text{lo+a}}$ and pA-Validation-$\texttt{PR}^{\text{fu}}$ we see no changes for the CPU time. For pA-Validation-

$\texttt{PR}^{\text{fu+a}}$ we see a few changes for the CPU time and for pA-Validation-$\texttt{PR}^{\text{gl}}$ and pA-Validation-$\texttt{PR}^{\text{gl+a}}$ the invariants-based predicates can sometimes be either helpful or disadvantageous.
It is noticeable that we see for pA-Validation-$\texttt{PR}^{\text{lo}}$ or pA-Validation-$\texttt{PR}^{\text{lo+a}}$ hardly changes when we compare it with pA-Verification and that we never can reduce the number of CEGAR refinements to zero.
A possible reason why we see no impact in our approach is that pA-Witnesses contain only invariants and no other precision relevant information that we can use for predicate precision initialization. Predicate analysis in CPAchecker can produce a program-precision file which contains the predicate precision. Reuse of program-precision files in CPAchecker has been confirmed to improve the performance [9]. However, when we translate the invariants from a pA-$\text{Witness}^{\text{no-true}^{+\text{h}}}$ into location predicates the precision information we receive is a subset from the precision information we get from a program-precision file for the same task. Both, the correctness witness and the precision file are created by using the precision from the ARG when the analysis finishes. For a precision file CPAchecker collects the precision information from the abstract states and for a correctness witness CPAchecker writes an invariant for a witness state by using the stored precision from an abstract state.
Another reason could be a bug in our approach or a bug in CPAchecker in general. We can reduce the number of refinements to zero for tasks only when we apply invariants-based predicates to the global scope (pA-Validation-$\texttt{PR}^{\text{gl}}$ and pA-Validation-$\texttt{PR}^{\text{gl+a}}$). Maybe we do not map the predicates to the correct location when we use the local scope.
Finally, we must conclude that our current approach of taking invariants from witnesses to initialize the predicate precision does not lead to an efficient precision reuse.

4.7 Validation of Correctness Witnesses with Witness Invariants as Additional Specification

4.7.1 Validating Witnesses produced by Predicate Analysis in CPAchecker

The bar chart in Fig. 4.5 illustrates the results we receive when validating *pA*-Witnesses. Each *pA*-Validation-`ISA` can reestablish the result for ≈97% of *pA*-Witnesses. The two tasks for which each *pA*-Validation-`ISA` detects a violation is due to a witness invariant violation (WIV) respectively (see Tab. 4.5). *k*-Induction performs also quite well and can validate ≈92% of *pA*-Witnesses. Ultimate Automizer has an accepting rate of ≈76%.

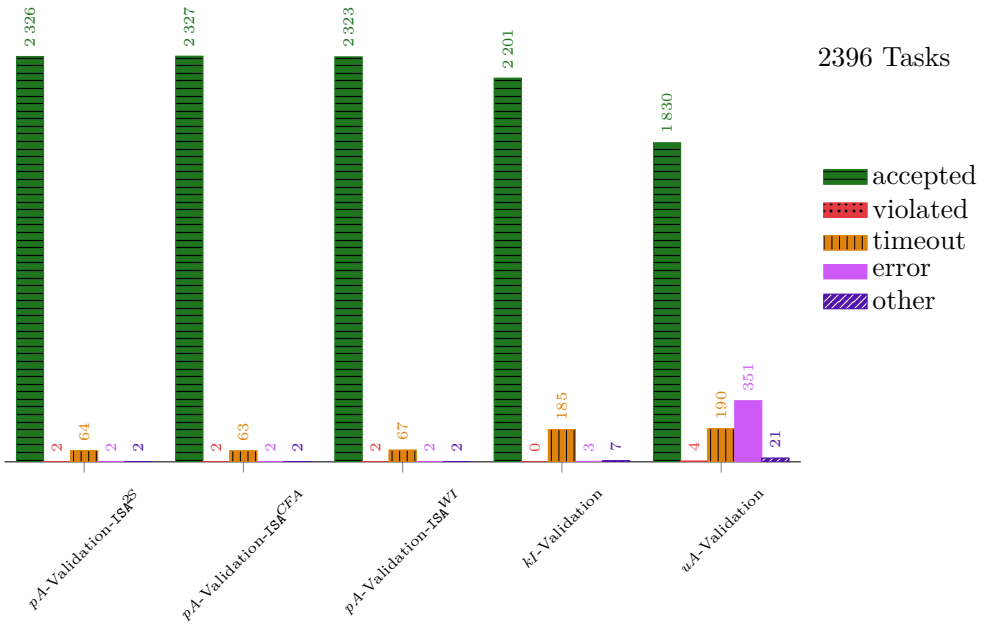

Figure 4.5: Validation results of *pA*-Witnesses

Table 4.5: Reason for *pA*-Witnesses violation for *pA*-Validation-`ISA`

configuration	*pA*-Validation-`ISA`^{2S}	*pA*-Validation-`ISA`^{CFA}	*pA*-Validation-`ISA`^{WI}
all violations	2	2	2
original specification	0	0	0
WIV	2	2	2

At first glance all validation approaches seem to perform well. However, in this diagram *true*-witnesses or *hidden-true*-witnesses are not filtered. Validating a *true*-witness or *hidden-true*-witness from *pA*-Witnesses is in particular trivial for *pA*-Validation-`ISA` because it produces an ISA without any invariant error states. When receiving a *true*-witness each *pA*-Validation-`ISA` is identical to predicate analysis for a verification task since the original safety property is the only specification.
In Fig. 4.6 we have filtered[8] all *true*-witnesses to receive *pA*-Witnesses$^{\text{no-true}}$. We do not filter *hidden-true*-witnesses in this section in general because

8. For 4 of 2396 *pA*-Witnesses we can not detect whether they are *non-trivial*-witnesses because each *pA*-Validation-`ISA` failed to produce statistics.

when we use the ISA^{WI} we can not detect *hidden-true*-witnesses (see section 6.2). In section 4.7.4 we will have a look how many *hidden-true*-witnesses are detected by comparing the printed statistics of *pA*-Validation-ISA^{2S} and *pA*-Validation-ISA^{CFA} with those of *pA*-Validation-ISA^{WI} to infer about *hidden-true*-witnesses.

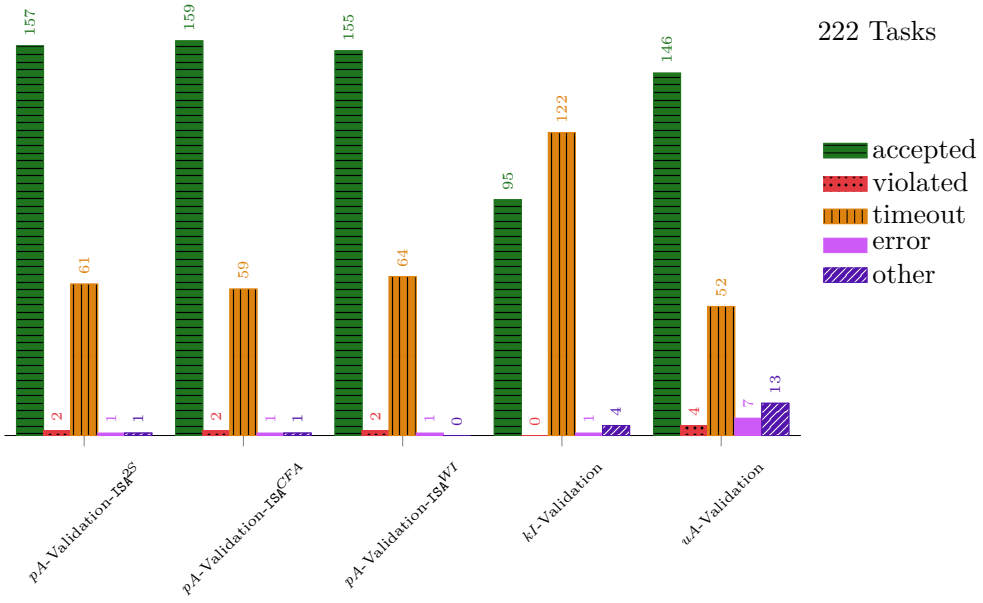

Figure 4.6: Validation results of *pA*-Witnesses$^{\text{no-true}}$

Regarding *pA*-Witnesses$^{\text{no-true}}$, *pA*-Validation-ISA^{2S} confirms ≈ 70%, *pA*-Validation-ISA^{CFA} confirms ≈72% and *pA*-Validation-ISA^{WI} confirms ≈70%. The accepting rate is lower compared to the accepting rate of *pA*-Witnesses because all approaches produce a noticeable high number of timeouts.
kI-Validation only accepts ≈43% of *pA*-Witnesses$^{\text{no-true}}$ and exceeds more often the time limit than it can accept *pA*-Witnesses$^{\text{no-true}}$. When *kI*-Validation is applied to tasks without loops, bounded model checking is sufficient and induction is not necessary. However, *pA*-Witnesses$^{\text{no-true}}$ are written in general for tasks that contain loops because *pA*-Verification finds invariants for loops. *kI*-Validation is configured with a static invariant generator which means *kI*-Validation can only use the *pA*-Invariants from *pA*-Witnesses$^{\text{no-true}}$ for an induction proof. It might be possible that the *pA*-Invariants give not enough information for *k*-induction when it is applied as validator for tasks with loops.
uA-Validation confirms ≈66% of *pA*-Witnesses$^{\text{no-true}}$ which is an accepting rate almost similar to *pA*-Validation-ISA.

Inspecting *pA*-Validation-ISA. We look at the following tasks in detail to inspect *pA*-Validation-ISA of *pA*-Witnesses:

#	Task	Subcategory
1	https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/loops/veris.c_NetBSD-libc__loop_true-unreach-call_true-termination.c	ReachSafety-Loops

2	https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/loop-invgen/id_build_true-unreach-call.i.v%2Blhb-reducer.c	ReachSafety-Loops
3	https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/loop-lit/cggmp2005b_true-unreach-call_true-termination.c	ReachSafety-Loops

Task #1 is one of the two tasks for which each *pA*-Validation-ISA produces a WIV. The reason for the rejection are problems in the intepretation of pointer arithmetics in the invariant. In the program of task #1 *pA*-Verification finds an invariant for a while loop. The program contains pointer variables and *pA*-Verification also writes those pointer values into the invariant. In the analysis *pA*-Verification receives address values for these pointers from the SMT solver which manages the pointer values internally. However, *pA*-Validation-ISA cannot prove the pointer values and therefore rejects the invariant. For task #1 *kI*-Validation returns unknown. *uA*-Validation detects a violation as well and we assume that it has also problems with the pointer values.

Task #2 is the second of the two tasks for which each *pA*-Validation-ISA produces a WIV. The correctness witness of task #2 contains a state labeled with an invariant that refers to the assignment of a program variable in a sequence of several program variable assignments. *pA*-Verification proofs that the invariant holds for a while loop that follows the sequence of assignments in the program. However, when using *pA*-Validation-ISA the witness state is mapped to the beginning of the sequence of assignments so that the invariant is applied to a CFA location where the invariant variables are not yet assigned. This wrong mapping is not due to an bug in the ISA. The witness is imprecise for our ISA approaches and the source-code guards of the witness state require this mapping.

We can see that *pA*-Validation-ISA produces sometimes a timeout for *pA*-Witnesses$^{\text{no-true}}$ in Fig. 4.6. But it seems that the invariants are not invalid since *pA*-Validation-ISA does almost never reject them. Nevertheless, *pA*-Validation-ISA has difficulties to proof the invariants within the time limit. We take a look at task #3 for which each *pA*-Validation-ISA exceeds the time limit. *pA*-Verification can label the task as correct-true. In the analysis of each *pA*-Validation-ISA we see that CEGAR can show that the path to the invariants-based error state is infeasible, but it can not refine the precision in a way that the invariant error state is not reached again in the next iteration. In each following iteration the reachability analysis enters the error state again and CEGAR checks that the state is unreachable and refines until the whole analysis exceeds the time limit. This problem might be similar for other tasks for which *pA*-Validation-ISA produces a timeout. When we count the CEGAR refinements for tasks for which *pA*-Verification can label the task as correct-true whereas *pA*-Validation-ISA produces a timeout we observe the following: *pA*-Verification often needs only 1 CEGAR refinement to proof the task whereas each *pA*-Validation-ISA

has made $\approx [50 - 5000]$ CEGAR refinements after it exceeds the time limit. We assume that the timeouts are not due to bugs in the implementation of an ISA. A reason might be the additional computation effort to validate the invariants.

Therefore, we take a closer look at the computation effort for validating pA-Witnesses$^{\text{no-true}}$. Fig. 4.7 compares the CPU time of pA-Validation-$\texttt{ISA}^{2S}$, pA-Validation-$\texttt{ISA}^{CFA}$, and pA-Validation-$\texttt{ISA}^{WI}$ with pA-Verification respectively. We can see that the CPU time increases for each pA-Validation-$\texttt{ISA}$ because of the additional work to verify invariants. Because of the invariants pA-Validation-$\texttt{ISA}$ has to do many refinements to exclude the reachability of invariants-based error states. This likely leads to more timeouts.

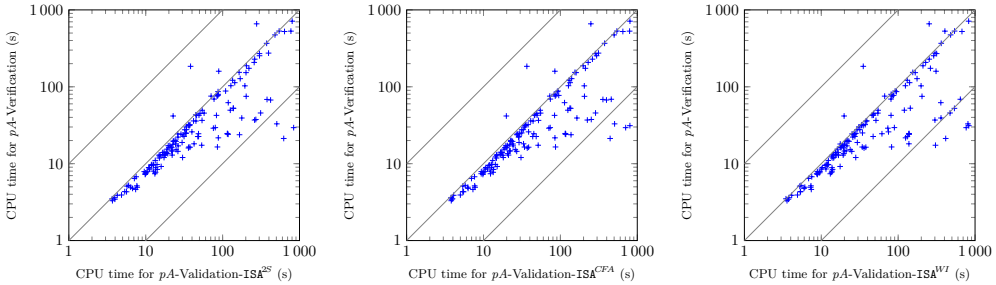

Figure 4.7: Scatter plots for comparing CPU time of pA-Validation-$\texttt{ISA}^{2S}$, pA-Validation-$\texttt{ISA}^{CFA}$ and pA-Validation-$\texttt{ISA}^{WI}$ with pA-Verification respectively for each task that both analyses are able to label as correct-true.

When we compare the results of pA-Validation-$\texttt{ISA}^{2S}$, pA-Validation-$\texttt{ISA}^{CFA}$ and pA-Validation-$\texttt{ISA}^{WI}$ we see that their results slightly differ. The difference between pA-Validation-$\texttt{ISA}^{2S}$ and pA-Validation-$\texttt{ISA}^{CFA}$ is marginal and only based on two tasks where pA-Validation-$\texttt{ISA}^{CFA}$ manages to not exceed the timeout and accepts the corresponding witnesses. When we compare pA-Validation-$\texttt{ISA}^{WI}$ with pA-Validation-$\texttt{ISA}^{CFA}$ or pA-Validation-$\texttt{ISA}^{2S}$ we see that pA-Validation-$\texttt{ISA}^{WI}$ exceeds the timeout for some tasks for which the preparatory step in pA-Validation-$\texttt{ISA}^{CFA}$ or pA-Validation-$\texttt{ISA}^{2S}$ fails due to recursion in these tasks.

4.7.2 Validating Witnesses produced by *k*-induction in CPAchecker

Fig. 4.8 illustrates the validation of kI-Witnesses. pA-Validation-$\texttt{ISA}^{2S}$ and pA-Validation-$\texttt{ISA}^{CFA}$ are able to confirm $\approx 52\%$ kI-Witnesses and pA-Validation-$\texttt{ISA}^{WI}$ can confirm $\approx 51\%$ of kI-Witnesses. They return a noticeable high number of violations mainly due to a WIV what we can see in Tab. 4.7. Also they produce a high number of errors which is mainly due to recursion errors. When we compare pA-Validation-$\texttt{ISA}^{2S}$ and pA-Validation-$\texttt{ISA}^{CFA}$ to pA-Validation-$\texttt{ISA}^{WI}$ we see that pA-Validation-$\texttt{ISA}^{WI}$ exceeds more often the time limit.

kI-Validation performs very well on validating kI-Witnesses. It accepts $\approx 92\%$ of kI-Witnesses and never rejects a witness.

uA-Validation confirms ≈ 62% and accepts therefore more kI-Witnesses than each pA-Validation-ISA which is noticeable since the production of kI-Witnesses and each pA-Validation-ISA are part of the same verification framework. But uA-Validation also rejects many kI-Witnesses.

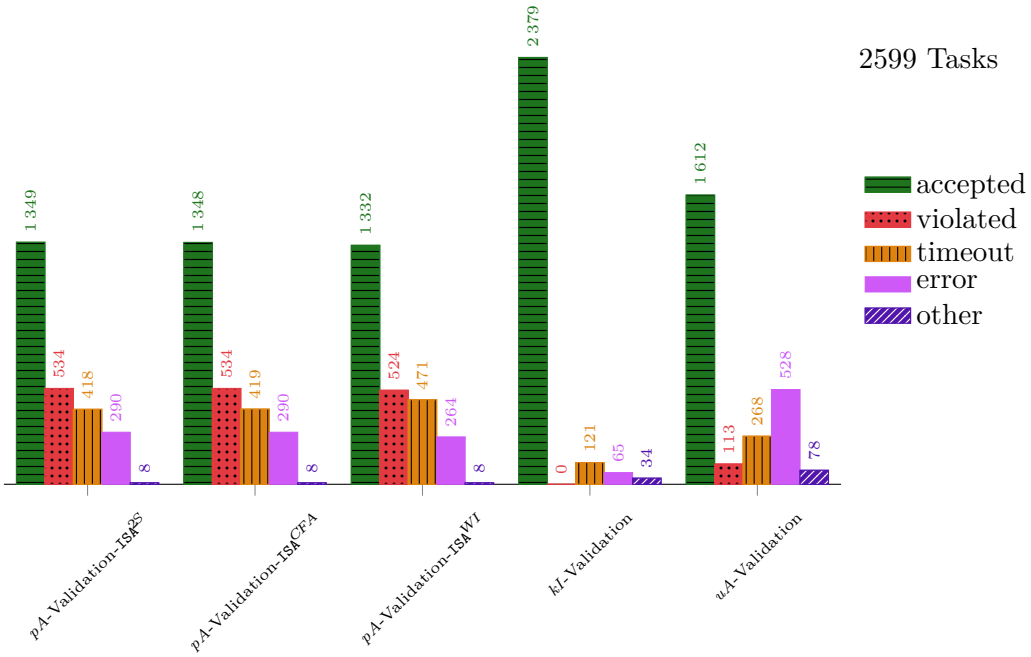

Figure 4.8: Validation results of kI-Witnesses

Table 4.7: Reason for kI-Witnesses violation for pA-Validation-ISA

configuration	pA-Validation-ISA^{2S}	pA-Validation-ISA^{CFA}	pA-Validation-ISA^{WI}
all violations	534	534	524
original specification	1	1	1
WIV	533	533	523

Fig. 4.9 illustrates the validation results of kI-Witnesses$^{\text{no-true}}$ [9]. pA-Validation-ISA^{CFA} and pA-Validation-ISA^{2S} accept ≈ 47% and pA-Validation-ISA^{WI} ≈ 46%. Each pA-Validation-ISA detects a property violation in more than 500 witnesses. For kI-Validation the acceptance rate is ≈ 95%. kI-Validation is mostly able to reestablish the original safety property with the invariants from kI-Witnesses$^{\text{no-true}}$. uA-Validation accepts ≈ 55% of kI-Witnesses$^{\text{no-true}}$.

9. For 292 of 2599 kI-Witnesses we can not detect whether they are *non-trivial*-witnesses because each pA-Validation-ISA approach failed to produce statistics.

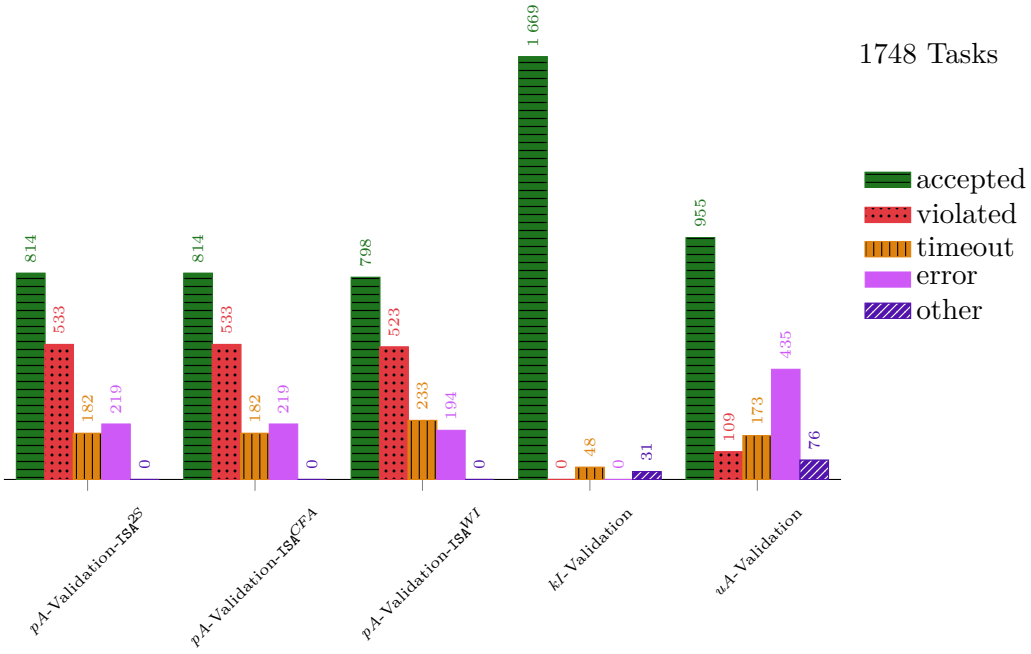

Figure 4.9: Validation results of kI-Witnesses$^{\text{no-true}}$

Each pA-Validation-`ISA` in CPAchecker and uA-Validation rejects a noticeable number of kI-Witnesses. We inspect this further to search for possible reasons.

Inspecting Correlation. We first have a look whether a correlation exists for rejections of kI-Witnesses$^{\text{no-true}}$ when we compare pA-Validation-`ISA` and uA-Validation. Therefore we look at all tasks for which at least one of the pA-Validation-`ISA` approaches observes a WIV. For a meaningful comparison we only regard tasks which are labeled by an independent uA-Verification run as correct-true. This gives us confidence that the rejection by uA-Validation is caused by invariants in kI-Witnesses$^{\text{no-true}}$ and not due to the original specification. Then we inspect for the reduced task set which status results uA-Validation returns. For uA-Validation we also present the rejection reason `unknown` which we have added before to the result `other` in Fig. 4.8 and Fig. 4.9. In Tab. 4.8 we can observe the results.

Table 4.8: Results for uA-Validation of kI-Witnesses$^{\text{no-true}}$ for which a pA-Validation-`ISA` detects a WIV and for which the corresponding programs are labeled correct-true by uA-Verification.

status	all	accepted	violated	timeout	error	unknown
uA-Validation	414	180	74	17	93	50

We can see that the acceptance rate of uA-Validation in Tab. 4.8 is $\approx$43%. It is smaller than the acceptance rate of $\approx$55% from Fig. 4.9. The violation rate increase from $\approx$ 6% in Fig. 4.9 to $\approx$ 18%. But this is not enough to see a tendency that uA-Validation returns a witness violation when pA-Validation-`ISA` does this as well. Moreover, uA-Validation is still able to confirm 180 kI-Witnesses$^{\text{no-true}}$ for which at least one pA-Validation-`ISA`

detects a WIV.
Now we observe a possible correlation the other way around. We look at kI-Witnesses$^{\text{no-true}}$ for which uA-Validation detects a violation and inspect what kind of result pA-Validation-`ISA` produces. Also here we take only those tasks into account that pA-Verification could verify in order to have confidence that a rejection is due to the invariants from kI-Witnesses$^{\text{no-true}}$.

Table 4.9: Results for pA-Validation-`ISA`^{2S}, pA-Validation-`ISA`^{CFA} and pA-Validation-`ISA`^{WI} of kI-Witnesses$^{\text{no-true}}$ for which an uA-Validation detects a violation and for which the corresponding programs are labeled correct-true by pA-Verification.

status	all	accepted	violated	timeout	error	other
pA-Validation-`ISA`^{2S}	109	10	82	0	17	0
pA-Validation-`ISA`^{CFA}	109	10	82	0	17	0
pA-Validation-`ISA`^{WI}	109	10	81	1	17	0

In Tab. 4.9 we see a correlation: When uA-Validation detects a violation in kI-Witnesses$^{\text{no-true}}$, pA-Validation-`ISA` mostly detects a violation as well. pA-Validation-`ISA` has only an accepting rate of $\approx$9%.

Inspecting pA-Validation-`ISA`. In order to find out why pA-Validation-`ISA` often detects a WIV we select some tasks and analyze pA-Validation-`ISA` for these tasks in detail. We discover two reasons: pA-Validation-`ISA` rejects kI-Witnesses$^{\text{no-true}}$ because the invariants in the witnesses are not valid or the witness is too imprecise for pA-Validation-`ISA` so that invariants are applied to locations where they not yet hold.

Invalid Invariants. For this reason we look at the following tasks:

#	Task	Subcategory
4	`https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/bitvector/num_conversion_1_true-unreach-call_true-no-overflow.c`	`ReachSafety-Bitvectors`
5	`https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/loops/sum03_true-unreach-call_false-termination.c`	`ReachSafety-Loops`

We first look at task #4. All pA-Validation-`ISA` approaches reject the corresponding correctness witness because of an WIV whereas uA-Validation and kI-Validation accept the witness. The program contains a while loop that is executed until the loop variable `unsigned char c` is equal to `8U`. As loop invariant that must hold at the head of the while loop kI-Verification has written a sequence of assignments over program variables in disjunctive form so that one of the operands should be satisfiable. However, the invariant is only satisifiable for the first iterations: The operands do not include assignments over variables when c is greater than 3. It seems that kI-Verification shows the safety property by induction without needing auxiliary invariants that correspond to the 4^{th}, 5^{th}, 6^{th} or 7^{th} iteration of the loop. Therefore, those auxiliary invariants lack as operands in the

resulting loop invariant. pA-Validation-ISA, however, checks the loop for the iteration steps greater than 3 and does not have the lacking operands. In consequence, pA-Validation-ISA shows that the negated invariant is satisfiable and rejects the witness. Since uA-Validation accepts the witness, we assume that uA-Validation does not verify the invariant for all iterations of the while loop.
For task #5 we encounter the same problem when pA-Validation-ISA analyzes the corresponding kI-Witness$^{\text{no-true}}$ and detects a WIV. The program contains an infinite while loop that never reaches the ERROR label in the while loop. kI-Verification can proof the unreachability and writes the loop invariant that is based on used auxiliary invariants into the witness. pA-Validation-ISA, however, cannot validate the invariant after a certain number of iterations of the while loop. This example also shows the limits for kI-Invariants. The loop is infinite and concrete assignments of program variables for a loop invariant can never represent the endless variable assignments. For a non-terminating while loop kI-Verification should write invariants with program variables in a relative context like $x = y$. Ultimate Automizer rejects the witness as well.

Imprecise Correctness Witness. For this reason we look at the following tasks:

#	Task	Subcategory
6	https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/float-benchs/filter1_true-unreach-call.c.p%2Bcfa-reducer.c	ReachSafety-Floats
7	https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/floats-esbmc-regression/Double_div_true-unreach-call.c	ReachSafety-Floats
8	https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/float-newlib/double_req_bl_1032d_true-unreach-call.c	ReachSafety-Floats
9	https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/loop-industry-pattern/mod3_true-unreach-call.c.v%2Bcfa-reducer.c	ReachSafety-Loops
10	https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/product-lines/email_spec0_product11_true-unreach-call_true-termination.cil.c	ReachSafety-ProductLines

We first inspect task #6. Each pA-Validation-ISA approach rejects the task because the corresponding kI-Witness$^{\text{no-true}}$ is too imprecise for the approach. The witness state that is labeled with an invariant is mapped to the beginning of a sequence of assignments in the program. However, the variable that is contained in the invariant does not yet exist at the beginning of the sequence and therefore pA-Validation-ISA immediately finds a WIV and rejects the witness. kI-Validation can validate the witness because it considers in general that a witness is imprecise [2] and maps the invariant to each location of the assignments. Since the invariant indeed holds after a certain program location in the sequence of assignments, kI-Validation can confirm the invariant. It is interesting that uA-Validation does not detect a violation of the invariant and produces a timeout. As the witness states can

be mapped to concrete CFA locations using program line and offset guards, *uA*-Validation should modify the CFA at the CFA location for which the invariant program variable is not yet assigned.
Task #7 is rejected by *pA*-Validation-`ISA` also because of an imprecise witness. When we extract the invariant from the witness state we apply this invariant one location to early where it does not yet hold. *kI*-Validation applies this invariant to both locations. In the analysis it will discard the invariant for the first location and proof it for the second location and is therefore able to validate the witness. *uA*-Validation confirms the witness. We encounter the same problem for #8, #9 and #10. *pA*-Validation-`ISA` can not validate the invariant because the invariant is applied to a location where it not yet holds. For all tasks *uA*-Validation also detects a violation. Each *pA*-Validation-`ISA` shows for all tasks a very small CPU time since it immediately rejects the witness when the invariant is mapped to the wrong program location. There are many other tasks for which *pA*-Validation-`ISA` rejects the witness because of a WIV with a very small CPU time and we assume that it is often because of an imprecise witness.

4.7.3 Validating Witnesses produced by Ultimate Automizer

In Fig. 4.10 we can see the validation results of *uA*-Witnesses.
pA-Validation-$\texttt{ISA}^{2S}$ and *pA*-Validation-$\texttt{ISA}^{CFA}$ are able to confirm ≈76% of *uA*-Witnesses whereas *pA*-Validation-$\texttt{ISA}^{WI}$ confirms ≈75%. The high amount of errors for all approaches is in particular due to recursion in some tasks. Violation of *uA*-Witnesses are because of WIVs and for one task because of the original specification (see Tab. 4.10).
kI-Validation confirms ≈85% of *uA*-Witnesses and never observes a witness violation.
uA-Validation accepts ≈88% of *uA*-Witnesses. However, it also produces the highest number for witness violations compared to *pA*-Validation-`ISA` and *kI*-Validation.

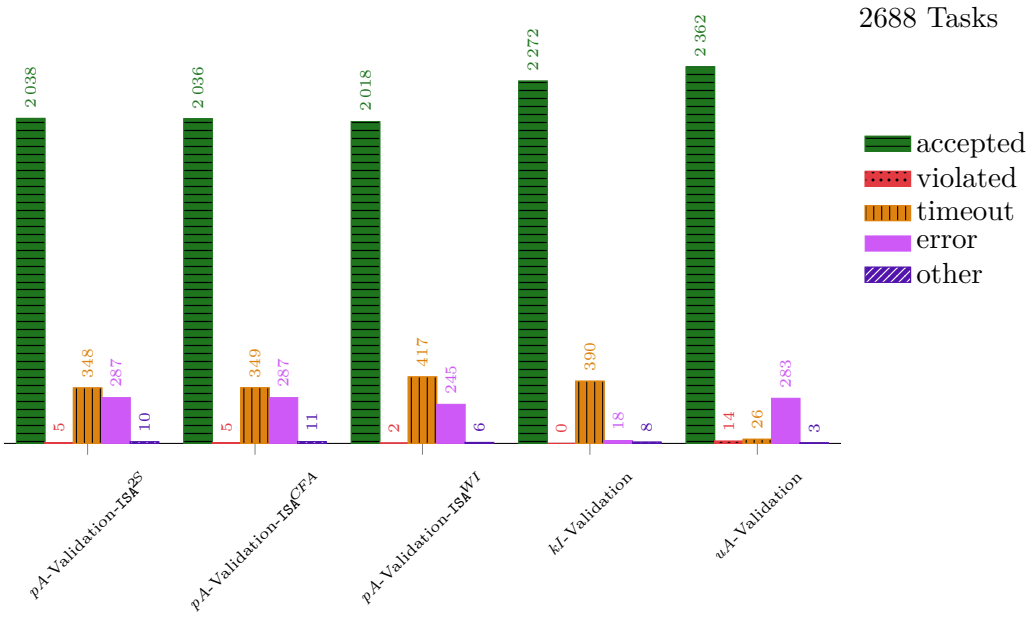

Figure 4.10: Validation results of uA-Witnesses

Table 4.10: Reasons for uA-Witnesses violation for pA-Validation-ISA

configuration	pA-Validation-ISA^{2S}	pA-Validation-ISA^{CFA}	pA-Validation-ISA^{WI}
all violations	5	5	2
original specification	1	1	1
WIV	4	4	1

Fig. 4.11 illustrates the validation results of uA-Witnesses$^{\text{no-true}}$. The reduction[10] leads to 248 uA-Witnesses$^{\text{no-true}}$.
We see in Fig. 4.11 that pA-Validation-ISA^{CFA}, pA-Validation-ISA^{2S} and pA-Validation-ISA^{WI} produce often errors for uA-Witnesses$^{\text{no-true}}$. The accepting rate of pA-Validation-ISA^{CFA}, pA-Validation-ISA^{2S} and pA-Validation-ISA^{WI} is $\approx$50%.
kI-Validation performs better than pA-Validation-ISA and can confirm $\approx$ 67%. However, it also exceeds the time limit for $\approx$ 30% of the uA-Witnesses$^{\text{no-true}}$. This indicates that the uA-Invariants give not enough information to support a safety proof in the induction.
uA-Validation performs very well on uA-Witnesses$^{\text{no-true}}$ and accepts $\approx$92%.

10. For 72 of 2688 uA-Witnesses we can not say whether they are *non-trivial*-witnesses because no pA-Validation-ISA approach is able to print statistics.

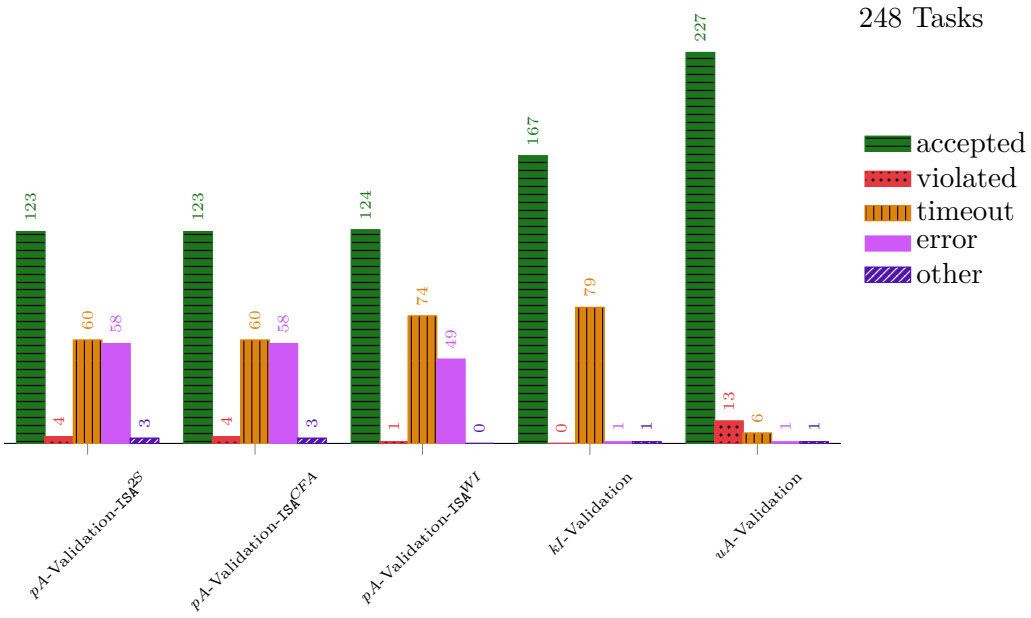

Figure 4.11: Validation results of uA-Witnesses$^{\text{no-true}}$

Inspecting pA-Validation: All pA-Validation-ISA approaches produce a high number of errors for uA-Witnesses$^{\text{no-true}}$. A possible reason is that uA-Verification can proof the specification for tasks for which predicate analysis in general has problems. Tab. 4.11 illustrates this. It contains the results for the programs of the 248 uA-Witnesses$^{\text{no-true}}$ when analyzed by an independent pA-Verification run. We see that pA-Verification does not perform well for those tasks and often produces an error or timeout. We see that the results highly correlate to the results of pA-Validation-ISA^{WI} in 4.11.

Table 4.11: Results of pA-Verification for tasks for which uA-Witnesses$^{\text{no-true}}$ are produced.

status	all	correct-true	correct-false	incorrect-true	incorrect-false	timeout	error	other
pA-Verification	248	127	0	0	0	68	49	4

It is interesting that pA-Validation-ISA^{CFA} and pA-Validation-ISA^{2S} reject four uA-Witnesses due to a WIV but pA-Validation-ISA^{WI} rejects only one uA-Witnesses due to a WIV. We inspect the following tasks:

#	Task	Subcategory
11	`https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/loops/invert_string_true-unreach-call_true-termination.c`	`ReachSafety-Loops`
12	`https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/psyco/psyco_security_true-unreach-call_false-termination.c`	`ReachSafety-ECA`
13	`https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/psyco/psyco_io_1_true-unreach-call_false-termination.c`	`ReachSafety-ECA`
14	`https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/psyco/psyco_accelerometer_1_true-unreach-call_false-termination.c`	`ReachSafety-ECA`

For task #11 each pA-Validation-ISA detects a WIV. The reason is that

pA-Validation-`ISA` can not replay the values of the pointer variables in the invariant. This problem is similar to an example we have shown in the section of validating pA-Witnesses. The loop invariant in the witness of #11 contains pointer variables of arrays. Ultimate Automizer, i.e. uA-Verification, assigns them to zero in the witness. pA-Validation-`ISA`, however, can not validate these pointer values and detects a WIV. uA-Validation also returns a violation of the witness and kI-Validation returns unknown.
For task #12, #13 and #14 pA-Validation-$\texttt{ISA}^{2S}$ and pA-Validation-$\texttt{ISA}^{CFA}$ detect a WIV wheras pA-Validation-$\texttt{ISA}^{WI}$ not. When we inspect #12 we can not see a bug in the ISAs that might explain the different behavior. The reason we observe is the structure of the correctness witness. The witness does not correspond to the loop structure of the program. Fig. 4.12 illustrates this witness. Witness state n_9 corresponds to the head of a while loop and is labeled with an invariant θ. In the corresponding program the while loop has several entering edges. However, n_9 has only two entering edges and one leaving edge: one edge comes from n_8, one edge is as self-transition and one edge leaves the state going to n_{10}. n_{10} has only a self-transition so the witness will stay forever in n_{10}. When we take the witness to build the ISA^{WI} we adopt the structure of the witness and hence create s_8, s_9 and s_{10} and the transitions with θ and $\neg\theta$ as assumptions. During the analysis θ is validated only once because as soon as the program leaves the loop head the ISA^{WI} proceeds to s_{10} and remains there forever. The ISA^{2S} and ISA^{CFA} differs from ISA^{WI}. They have an automaton state for n_9 that can be revisited. But when the program reenters the while loop, the invariant is not satisfiable and therefore pA-Validation-$\texttt{ISA}^{CFA}$ or pA-Validation-$\texttt{ISA}^{2S}$ will take the transition to the error state assuming the negated invariant.
However, we determine that this WIV happens because ISA^{CFA} and ISA^{2S} overapproximate the invariant based error states and this overapproximation leads to false alarms. The witness might not correspond to the loop structure but this should not be a reason to reject the witness. Let θ be the invariant at n_9: When we build the ISA^{CFA} or ISA^{2S} based on the correctness witness from #12 we apply a reachability analysis on the witness first to get the location invariants. When we iterate over the reached set we get θ but also *true* for the same CFA location. For soundness we use at this location the conjunction of θ and *true* which is equal to θ. In consequence, θ must always be verified when the loop head is entered. However, when the program *reenters* the loop head the original witness semantics specify that there is no invariant to proof since the original witness remains in n_{10}. We see that our implementations of pA-Validation-$\texttt{ISA}^{CFA}$ and pA-Validation-$\texttt{ISA}^{2S}$ are not precise because they overapproximate the error states.

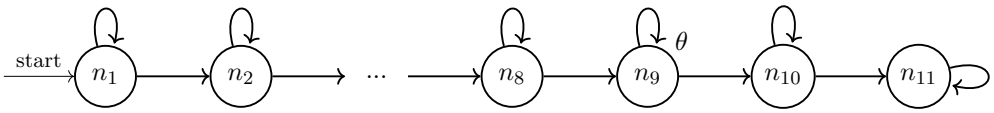

Figure 4.12: Correctness witness produced by Ultimate Automizer for task #12 with a *non-trivial*-Invariant θ at n_9

The correctness witnesses for tasks #13 and #14 trigger the same problem. Their structure is similar to the structure of the witness of #11 in Fig. 4.12 what leads again to the different rejection behavior of *pA*-Validation-$\texttt{ISA}^{CFA}$ and *pA*-Validation-$\texttt{ISA}^{2S}$ compared to *pA*-Validation-$\texttt{ISA}^{WI}$.

4.7.4 Comparing the Detection of Correctness Witness Types

Although *pA*-Validation-$\texttt{ISA}^{WI}$ can not detect *hidden-true*-witnesses we can check for *pA*-Witnesses, *kI*-Witnesses and *uA*-Witnesses whether they contain *hidden-true*-witnesses. Our check is based on the following detection: In *pA*-Validation-$\texttt{ISA}^{WI}$ the witness is a *non-trivial*-witness whereas in *pA*-Validation-$\texttt{ISA}^{CFA}$ and *pA*-Validation-$\texttt{ISA}^{2S}$ the witness is either a *true*-witness or *hidden-true*-witness. Then the witness is a *hidden-true*-witness because *pA*-Validation-$\texttt{ISA}^{WI}$ is able to exclude *true*-witnesses.
Tab. 4.12 shows the detection of correctness witness types. We can see that *pA*-Verification and *uA*-Verification produce no *hidden-true*-witnesses because the numbers are all equal. However, for *uA*-Verification we can estimate that 147 *uA*-Witnesses are *hidden-true*-witnesses because they are detected as *hidden-true*-witnesses or *true*-witnesses by *pA*-Validation-$\texttt{ISA}^{CFA}$ and *pA*-Validation-$\texttt{ISA}^{2S}$ but detected as *non-trivial*-witnesses by *pA*-Validation-$\texttt{ISA}^{WI}$.

Table 4.12: Comparing the detection of correctness witness types for *pA*-Validation-ISA approaches. Only tasks are considered for which all three *pA*-Validation-ISA approaches can produce statistics.

	approach	*non-trivial*-witnesses	*hidden-true*-witnesses or *true*-witnesses	*true*-witnesses
2396 *pA*-Witnesses	*pA*-Validation-$\texttt{ISA}^{2S}$	220	2169	-
	pA-Validation-$\texttt{ISA}^{CFA}$	220	2169	-
	pA-Validation-$\texttt{ISA}^{WI}$	220	-	2169
2599 *kI*-Witnesses	*pA*-Validation-$\texttt{ISA}^{2S}$	1696	559	-
	pA-Validation-$\texttt{ISA}^{CFA}$	1696	559	-
	pA-Validation-$\texttt{ISA}^{WI}$	1696	-	559
2688 *uA*-Witnesses	*pA*-Validation-$\texttt{ISA}^{2S}$	40	2337	-
	pA-Validation-$\texttt{ISA}^{CFA}$	40	2337	-
	pA-Validation-$\texttt{ISA}^{WI}$	187	-	2190

To search for explanations why we receive *hidden-true*-witnesses from *uA*-Verification in our implementations we look at the following tasks:

#	Task	Subcategory
15	https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/loop-invariants/eq1_true-unreach-call_true-valid-memsafety_true-no-overflow_false-termination.c	ReachSafety-Loops
16	https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/ssh-simplified/s3_srvr_1b_true-unreach-call_false-termination.cil.c	ReachSafety-ControlFlow
17	https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/ldv-regression/test28_true-unreach-call_true-termination.c	ReachSafety-Heap
18	https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/bitvector-regression/implicitunsignedconversion_true-unreach-call_true-termination.c	ReachSafety-BitVectors

In task #15 we inspect that the correctness witness produced by *uA*-Verification contains transitions which can not be taken in the analysis in CPAchecker. For example, the assignment `unsigned int w = __VERIFIER_nondet_uint();` is represented by two states and two transitions: one transition with the whole assignment as guard that goes into the first state and a second transition that goes into the successor state with `__VERIFIER_nondet_uint();` as guard. In the analysis of CPAchecker when the witness automaton stays in the first state the transition to the successor state can not be taken since CPAchecker applies already the next program operation. In consequence, the automaton remains in the first state.
In task #16 and #17 we encounter the same problem. The program contains calls of `VERIFIER_nondet_uint();` and the correctness witness produced by *uA*-Verification has two states when this function is called in the program.
In task #18 the witness from *uA*-Verification has one state that is labeled with invariant 0. We assume that *uA*-Verification explicitly labels the state with invariant 0 to note the unreachability of the state because the state is indeed unreachable when we look at the program location that refers to the state. Nevertheless, this leads to an empty set of location invariants for *pA*-Validation-$\texttt{ISA}^{2S}$ and *pA*-Validation-$\texttt{ISA}^{CFA}$ and is therefore classified as *hidden-true*-witness in our implementations.
In conclusion, we have found two reasons why we detect *hidden-true*-witnesses in *uA*-Witnesses: Firstly, Ultimate Automizer writes two states for the call `VERIFIER_nondet_uint();` in the witness. This has as consequence that in the analysis in CPAchecker the automaton stucks in the first state. Secondly, an *uA*-Witness might contain an invariant 0 to label explicitly an unreachable state.

4.7.5 Summary

Each ISA approaches can be used to apply witness invariants as additional verification goal. However, only the ISA^{WI} is precise whereas the ISA^{CFA} and ISA^{CFA} can produce false alarms because they overapproximate the error states. Moreover, the ISA^{WI} in CPAchecker does not need a preparatory step and requires less code in CPAchecker. The ISA^{WI} also produces less

error results. We can inspect this in all validation results diagrams since the preparatory step for pA-Validation-$\texttt{ISA}^{2S}$ and pA-Validation-$\texttt{ISA}^{CFA}$ sometimes fails due to a recursion error. However, for these tasks pA-Validation-$\texttt{ISA}^{WI}$ produces a timeout.
The main advantage of ISA^{2S} and ISA^{CFA} is the compact automaton structure. In Fig. 4.13 pA-Validation-$\texttt{ISA}^{2S}$ and pA-Validation-$\texttt{ISA}^{CFA}$ show a lower computation time compared to pA-Validation-$\texttt{ISA}^{WI}$ in particular for kI-Witnesses$^{\text{no-true}}$. For pA-Witnesses$^{\text{no-true}}$ and uA-Witnesses$^{\text{no-true}}$ they also perform for a few tasks better compared to pA-Validation-$\texttt{ISA}^{WI}$.
Tasks for which a verifier produces huge correctness witnesses are sometimes difficult for pA-Validation-$\texttt{ISA}^{WI}$. For these witnesses pA-Validation-$\texttt{ISA}^{WI}$ exceeds more often timeouts compared to pA-Validation-$\texttt{ISA}^{2S}$ and pA-Validation-$\texttt{ISA}^{WI}$. The reason we assume is the continuous matching of source-code guards during the analysis which is required by the ISA^{WI}.

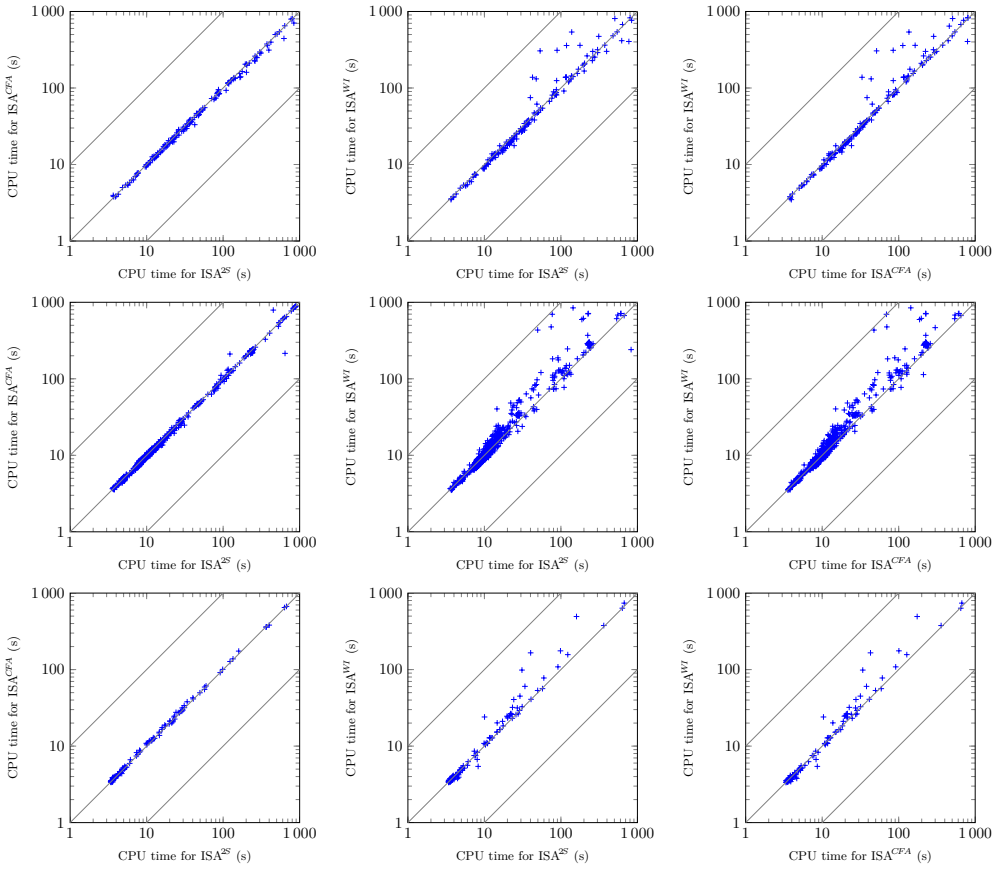

Figure 4.13: Scatter plots for comparing CPU time of our pA-Validation-$\texttt{ISA}$ approaches respectively for tasks for which both analyses accepted the correctness witness. **Upper Row:** Comparing pA-Validation-$\texttt{ISA}$ approaches for pA-Witnesses$^{\text{no-true}}$. **Middle Row:** Comparing pA-Validation-$\texttt{ISA}$ approaches for kI-Witnesses$^{\text{no-true}}$. **Lower Row:** Comparing pA-Validation-$\texttt{ISA}$ approaches for uA-Witnesses$^{\text{no-true}}$.

We have seen that pA-Validation-$\texttt{ISA}$ is able to confirm correctness witnesses from uA-Verification and kI-Verification. However, we have also seen problems which have lead to WIVs in pA-Validation-$\texttt{ISA}$. Imprecise correctness witnesses can not be handled by our ISA implementations. Invariants in kI-Witnesses for a loop head might not reflect all iterations. Pointer

variables with value assingments in witness invariants can be violated in the validation.
Furthermore, we have discovered that *uA*-Witnesses can be *hidden-true*-witnesses in the context of a witness analysis in CPAchecker.

Chapter 5

Conclusion

We implemented several approaches to validate correctness witnesses using predicate analysis. In a large benchmark study we evaluated whether predicate analysis can understand its own and other witnesses and whether it can reestablish the result. We have shown that predicate analysis in CPAchecker can be applied as validator for correctness witnesses in CPAchecker.
In our first validation approach we investigated in particular the precision reuse of invariants from correctness witnesses produced by predicate analysis. We have seen that the predicate analysis-based validator can verify for almost all tasks the original specification. However, precision reuse with witness invariants does not decrease computation time in our approach. Moreover, applying the invariant-based predicate to the exact location does not decrease CEGAR refinements. We conclude that the information from the invariants might not be sufficient to create a initial predicate precision that decreases the computation time or we have a bug in the precision mapping in CPAchecker. Further investigations whether we have a bug might be helpful.
Each invariants specification automata (ISA) we have implemented in CPAchecker allows us to validate the original specification and to validate the correctness witness invariants. We have seen that predicate analysis can validate the majority of correctness witnesses produced by predicate analysis as well. Furthermore, predicate analysis validation understands correctness witnesses produced by k-induction in CPAchecker or produced by automata-based verification in Ultimate Automizer.
We propose that correctness witnesses should be precise. For our approaches imprecise correctness witnesses can not be handled and trigger a WIV. The source-code guards of a transition that enters a witness state with an invariant should correspond to the program location where the invariant indeed holds. Moreover, correctness witnesses and their invariants should reflect the program structure. A witness with a loop invariant that needs to be verified only once contradicts our understanding of validating a loop invariant. We have seen that validators have problems with pointer variables in invariants. The reason might be due to bugs which should be fixed or that a formal standard lacks how pointer values are written

into correctness witnesses. We have also seen that correctness witnesses by Ultimate Automizer can be *hidden-true*-witnesses in our implementations in CPAchecker. One reason is that CPAchecker can not understand the transitions. This behavior might be due to a bug or it lacks a more precise formalization for transitions.
The ISA^{WI} is the only ISA that defines the witness invariants as additional verification goal and guarantees to reflect the semantics of the correctness witness. Since ISA^{CFA} and ISA^{2S} overapproximate invariants-based error states, they can produce false alarms in the analysis.

5.1 Future Work

The invariants we get from the correctness witnesses could be negated so that we can investigate, whether predicate analysis as validator can reject correctness witnesses when the invariants are intentionally wrong.

Each ISA is independent from the applied abstraction-technique. This means, that other techniques in CPAchecker can validate theoretically the original specification and the witness invariants of correctness witnesses. This might be interesting for a developer of CPAchecker since the developer gets the opportunity to validate correctness witnesses with the analysis he or she has implemented. For the community of software verification using an ISA can be beneficial because validating the correctness witnesses of a program with different techniques in CPAchecker can increase the trustworthiness in the program when the result is reestablished.

Our approach of validating correctness witness invariants could be combined with the program generation concept presented in [12][13][15]. According to this concept a program P that has been verified for a given specification can be transformed into a behaviorally equivalent program P' which can be verified more efficient. The abstract states of the ARG can be used to create P'.
When we validate the correctness witness of a program and use the ISA^{WI} the invariant error states in the ISA^{WI} extend the specification and affect the construction and structure of the ARG. When the validation task finishes and we build a program using the ARG the resulting program also includes the invariant error states in form of code. This program can be analyzed by a verifier and a validator is not necessary.

Chapter 6

Appendix

6.1 Receiving a set of location invariants from a correctness witness

The prepatory step to receive the set of location invariants from a correctness witness has the following substeps in CPAchecker: (I) Parsing a correctness witness into a correctness witness automaton. (II) Performing a rechability analysis on the witness automaton.(III) Extracting the location invariants from the reached set. For (I) and (II) implementations for k-induction validation in CPAchecker already exists and we can reuse these implementations.

Parsing a Correctness Witness. For an analysis we can parse the correctness witness into an *observer automaton* that can be understood by CPAchecker. The automaton (witness automaton) is a protocol automaton $A = (S, \Sigma, \delta, s_{init}, F)$ where S denotes the set of automaton states, s_{init} denotes the starting state, Σ denotes the alphabet over the source-code guards, $\delta \subseteq (S \times \Sigma \times S)$ denotes the set of automaton transitions and F denotes the set of accepting states. The structure refers to the structure of the correctness witness. For each correctness witness state n we built an automaton state s. For each witness transition $(n, (guards), n')$ we built an automaton transition $(s, \texttt{match}(guards), s')$. The function `match` evaluates whether a CFA edge matches with the source-code guards and returns *true* or *false*. When we unroll the CFA the CFA edges must satisfy the guards. If a CFA edge does not match with the guards the automaton remains in the current state using the self-transition. For each witness target state n' that is labeled with a *non-trivial*-Invariant we store the invariant in the parsed automaton transition that leads to s' which is the corresponding state of n' in the parsed automaton. If the witness target state has no invariant we store *true* as invariant in the transition.
Fig 6.1 illustrates a graph of the parsed witness automaton of 2.2. For instance, the automaton enters state s_2 using the transition (s_1, `match(PL=3, enterLoopHead)`, s_2), if the CFA edge goes into the head of the while loop. Moreover, the transition stores the invariant $x = y$ because the original

witness target state s_2 is labeled with invariant $x = y$.

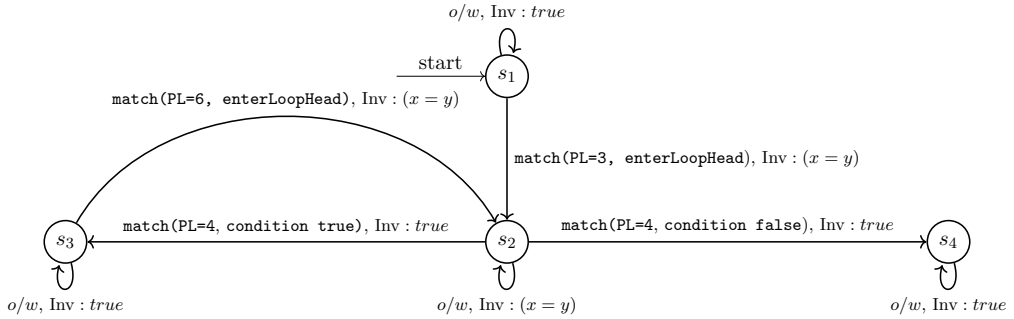

Figure 6.1: Parsed witness automaton in CPAchecker of the correctness witness from Fig. 2.2

Reachability Analysis on a Witness Automaton. We can perform a reachability analysis on the witness automaton which we have parsed before. In order to do so we embed the witness automaton into an *Automaton CPA*. Together with a *Location CPA* we can add them as component CPAs to a *Composite CPA* and perform a reachability analysis. Note that we do not specify any safety properties. The analysis terminates when all reachable states have been processed ($\widehat{=}$ the waitlist is empty). During the analysis, automaton states of the *Automaton CPA* are discovered based on the witness automaton transitions. This means in particular that we get a successor automaton state with a *non-trivial*-Invariant when the underlying witness automaton transition stores a *non-trivial*-Invariant. When the ARG construction is finished we obtain a set of reachable abstract states and each abstract state contains an automaton state.

Extracting the Invariants. After the witness is analyzed we receive a reachable set of composite abstract states. We can iterate over this set to get the invariants and to get for each invariant its corresponding CFA location[1]. For each composite abstract state we extract the CFA location which we get from the *Location CPA* and extract its witness automaton state from the *Automaton CPA*. We can inspect for each automaton state whether it has a *non-trivial*-Invariant. The invariant *true* is always valid and therefore ignored. We store each *non-trivial*-Invariant together with the location as location invariant (l, θ). Each location invariant is added to the set of location invariants I. If we receive two or more invariants for the same CFA location we use the conjunction for the invariants. After iterating over the set of composite abstract states we can use I for our predicate analysis validation approaches. If I is empty the correctness witness is a *true*-witness or *hidden-true*-witness.

1. For validation with k-induction an invariant is mapped to a group of CFA locations so that the invariant might be applied to several CFA locations.

6.2 Statistics Features

Detecting Types of Correctness Witnesses. For our evaluation we distinguish between correctness witnesses that give us at least one *non-trivial*-Invariant (*non-trivial*-witnesses) and those which do not contain any *non-trivial*-Invariants (*true*-witnesses) or for which states with *non-trivial*-Invariants are unreachable (*hidden-true*-witnesses).
For our approach to initialize the predicate precision with correctness witness invariants we count the number of location invariants which we receive after the preparatory step. We add this number to the statistics of each task. If the number is zero, we infer that the witness is a *true*-witness or *hidden-true*-witness.
Counting the number of location invariants does not work for the ISA^{WI}. Therefore, we follow a different approach for an ISA in general. When we validate correctness witness invariants by using an ISA we count the number of states that contain transitions with assumptions. We add this number to the statistics output of each task. A transition with an assumption can only be based on an invariant in our ISAs implementations.
The ISAs differ in their detection behavior. An ISA^{2S} or ISA^{CFA} that has no states with transitions with assumptions is based on a *true*-witness or *hidden-true*-witness. A *hidden-true*-witness or *true*-witness leads to an empty set of location invariants when we built an ISA^{2S} or an ISA^{CFA}. An ISA^{WI} that has no states with transitions with assumptions is always based on a *true*-witness. In our implementation we can not detect *hidden-true*-witnesses when we use the ISA^{WI} because unreachable states with invariants are included as unreachable ISA^{WI} states. Therefore, the ISA^{WI} has states with transitions with assumptions.
We must clarify that CPAchecker might break the analysis without delivering statistics information. If this is the case we can not infer about the existence of *non-trivial*-Invariants. For instance, the parsing of the witness can lead to an error or the reachability analysis we apply in the preparatory step to get location invariants can fail. If we get no statistics for a validation task we can not say whether the correctness witness is a *non-trivial*-witness.
In general, the detection of *non-trivial*-witnesses depends on our implementations in CPAchecker. It might be possible that we wrongly classify a *non-trivial*-witness as *hidden-true*-witness or *true*-witness. Since we can not guarantee to detect all *non-trivial*-witnesses, the set of *non-trivial*-witnesses which we can detect is a subset of the real set of *non-trivial*-witnesses.

Detecting Witness Invariant Violation (WIV). If predicate analysis validation with an ISA in CPAchecker rejects a witness we explicitly output in the statistics whether it is due to an invariant violation.

6.3 Specific Settings in CPAchecker

For our approaches explained in chapter 3 we need to set configurations in CPAchecker. The configurations are summarized in Tab. 6.1.

Table 6.1: Configurations in CPAchecker for precision reuse of witness invariants and for building an ISA

Option	Values	Meaning
analysis .validateCorrectnessWitness	NONE, INVARIANTS AUTOMATON, LOCATION INVARIANTS AUTOMATON, WITNESS AUTOMATON	Add invariants as additional specification by using an invariant specification automaton INVARIANTSAUTOMATON: section 3.3.1 LOCATIONINVARIANTSAUTOMATON: section 3.3.2 WITNESSAUTOMATON: section 3.3.3
cpa.predicate .correctnessWitness .reuseInvariants	false, true	Reuse invariants from the correctness witness by transforming them into predicates and adding the predicates into the initial predicate precision
cpa.predicate .correctnessWitness .atomPredicatesFromFormula	false, true	Witness invariants are transformed into atomic predicates
cpa.predicate .refinement .splitItpAtoms	false, true	Needs to be enabled in order to get atomic predicates
cpa.predicate .correctnessWitness .witnessInvariantScope	LOCATION, FUNCTION, GLOBAL	Defines the precision scope for the invariants-based predicates - LOCATION: predicates applied at the corresponding location - FUNCTION: predicates applied at all locations in the function scope - GLOBAL: predicate applied at all program locations

Bibliography

[1] D. Beyer. "Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016)". In: *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016, Eindhoven, The Netherlands, April 2-8)*. LNCS 9636. Springer-Verlag, Heidelberg, 2016, pp. 887–904.

[2] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. "Correctness Witnesses: Exchanging Verification Results Between Verifiers". In: *Proc. FSE*. ACM, 2016, pp. 326–337.

[3] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. "Witness Validation and Stepwise Testification Across Software Verifiers". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, 2015, pp. 721–733.

[4] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. "Witness Validation and Stepwise Testification across Software Verifiers". In: *Proc. FSE*. ACM, 2015, pp. 721–733.

[5] D. Beyer, M. Dangl, T. Lemberger, and M. Tautschnig. "Tests from Witnesses: Execution-Based Validation of Verification Results". In: *Proceedings of the 12th International Conference on Tests and Proofs (TAP 2018, Toulouse, France, June 27-29)*. Ed. by C. Dubois and B. Wolff. LNCS 10889. Springer, 2018, pp. 3–23.

[6] D. Beyer, M. Dangl, and P. Wendler. "A Unifying View on SMT-Based Software Verification". In: *J. Autom. Reasoning* 60.3 (2018), pp. 299–335.

[7] D. Beyer, M. Dangl, and P. Wendler. "Boosting k-Induction with Continuously-Refined Invariants". In: *Proceedings of the 27th International Conference on Computer Aided Verification (CAV 2015, San Francisco, CA, USA, July 18-24)*. LNCS 9206. Springer-Verlag, Heidelberg, 2015, pp. 622–640.

[8] D. Beyer, M. E. Keremoglu, and P. Wendler. "Predicate Abstraction with Adjustable-Block Encoding". In: *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2010, Lugano, October 20-23)*. FMCAD, 2010, pp. 189–197.

[9] D. Beyer, S.Löwe, E.Novikov, A.Stahlbauer, and P.Wendler. "Precision reuse for efficient regression verification". In: *Proc. FSE*. ACM, 2013, pp. 389–399.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification*. Springer Berlin Heidelberg, 2000, pp. 154–169.

[11] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. “Lazy Abstraction”. In: *SIGPLAN Not.* 37.1 (Jan. 2002), pp. 58–70.

[12] M.-C. Jakobs and H. Wehrheim. “Programs from Proofs of Predicated Dataflow Analyses”. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. SAC ’15. Salamanca, Spain: ACM, 2015, pp. 1729–1736.

[13] M.-C. Jakobs and H. Wehrheim. “Programs from Proofs: A Framework for the Safe Execution of Untrusted Software”. In: *ACM Trans. Program. Lang. Syst.* 39.2 (Mar. 2017), 7:1–7:56.

[14] R. Jhala and R. Majumdar. “Software Model Checking”. In: *ACM Comput. Surv.* 41.4 (Oct. 2009), 21:1–21:54.

[15] D. Wonisch, A. Schremmer, and H. Wehrheim. “Programs from Proofs – A PCC Alternative”. In: *Computer Aided Verification*. Ed. by N. Sharygina and H. Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 912–927.