University of Passau
Department of Informatics and Mathematics

Bachelor Thesis

Automata-Guided Synthesis and Reuse of Precisions

Author:
Sebastian Böhm

September 28, 2017

Advisors:
Prof. Dr. Sven Apel
Andreas Stahlbauer
(Chair of Software Engineering I)

Sebastian Böhm:
Automata-Guided Synthesis and Reuse of Precisions
Bachelor Thesis, University of Passau, 2017

Abstract

In software verification based on counterexample guided abstraction refinement (CEGAR) reducing the number of refinement iterations is desired in order to lower the costs of the verification algorithm. The general approach to achieve this is to provide the algorithm with additional precisions before a spurious counterexample is found and a refinement is needed. Ideally, the resulting abstract model is precise enough to contain less spurious counterexample paths.

In this work we discuss some shortcomings of precision reuse, an existing approach to reduce the number of refinements, by demonstrating them on simple examples. We find that its main problem is that the reused information is inherently static and cannot adapt to new verification tasks. With this insight we develop a new technique that synthesizes new precisions from symbolic automata on-the-fly by observing the creation and exploration of the abstract model and suggesting new precisions in certain situations. The flexibility of this approach can be further improved by allowing templates for the generation of precisions. Our method is defined as a configurable program analysis (CPA) and therefore, it can be easily integrated into existing analyses.

We show how specifications can be transformed into precision mining automata that synthesize precisions crucial for the separation of violating and non-violating program paths. This work also provides an implementation of this technique as well as an experimental evaluation to demonstrate its efficiency and effectiveness. Our experiments confirm, that precision mining from specifications can indeed severely reduce the number of refinements.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The complexity of today's software systems calls for means to reason about their correctness. While testing provides a practical solution to give hints about whether programs are working as intended, especially for safety-critical applications one would like to be able to prove that software is error-free. This is what software verification tries to accomplish. Software verification takes a program and some specification as input and proves in finite time if the program violates the given specification or not. Since in general this problem is undecidable and even under restrictions still is computationally expensive, a lot of effort has been put into applying software verification to real-world applications.

Counterexample-guided abstraction refinement (CEGAR) is a verification algorithm that uses abstraction to cope with state-space explosion and automatically adjusts the level of abstraction until the result of the analysis is sound. This is done by iteratively refining an initially coarse abstract model of the program until it is precise enough for the analysis either to find a counterexample that is not an artifact of the abstract model or to prove it save. Those refinements however, are computational expensive and thus, it is desired to reduce their number to improve the overall performance of the verification task.

Goal The goal of this thesis is to define a new method to be used with CEGAR-based verification algorithms that *automatically refines the precision* of the abstract model *on-the-fly* in order to reduce the amount of refinements needed to produce a sound result by using *sumbolic automata* and *templates*. In order to demonstrate the benefit of our technique we provide an implementation that gets investigated in an experimental evaluation.

Related Work Over the years different approaches on how to choose or generate precisions have been proposed: In the work that first introduced predicate abstraction, predicates are chosen from the guards of the transition system of the program and more predicates are computed from them [10]. With CEGAR the search for precisions can be automated by performing a Craig-interpolation of the path formula of a spurious counterexample [9, 11]. This technique exploits the observation that the reason for an abstract counterexample being infeasible is encoded in the proof that it is infeasible (i.e. in the trace of the counterexample). When the analysis discovers a counterexample it is checked, whether it is valid or just an artifact of a too imprecise model. If the counterexample indeed is spurious, new predicates are extracted from it via Craig-interpolation and the refined model now does not contain that counterexample anymore.

Precision reuse recycles the precision of an earlier verification run to make the initial abstract model precise enough to rule out spurious counterexamples thus, saving refinements [8]. The success of this technique depends on the similarity of the program under verification to the program the reused precision originates from and on whether the used specification is the same. Relying on this type of information is suitable for situations like regression verification but it does not adapt well to new verification problems.

Contribution In this work we make the following contributions:

- We investigate situations where *precision reuse* does not work well or even has *negative effects* on performance. Based on simple examples we explain the reasons for the behavior of this technique in the observed situations.
- We introduce a new technique that *generates precisions* during the verification run so that the abstract model is precise enough to rule out some spurious counterexamples and thus, save refinements. Therefore, we define a CPA that uses *symbolic automata* and *templates* to generate precisions *on-the-fly*.
- We provide an *implementation* of this technique which uses specifications as precision mining automata and investigate its effectiveness and efficiency in an *experimental evaluation*.

Overview After providing some background knowledge about the used terms and concepts in the first chapter, we revisit precision reuse [8], an existing method that supplies already computed results to future verification tasks in order to reduce refinement iterations. While a detailed experimental study shows that precision reuse can cause a huge speedup when successive revisions of a program are verified [8], we found scenarios where this method does not work well or even worsens performance. By investigating these situations we found out that they cannot always be easily circumvented.

The idea behind precision reuse is to provide the analysis with additional precisions that make the abstract model precise enough to rule out spurious counterexamples before they would be discovered, and therefore refinement iterations of the CEGAR-loop can be saved. In chapter four, we describe a new method based on this idea that utilizes symbolic automata to generate new precisions on-the-fly. That way our technique can be applied to new verification problems without the need for previously computed results. We take advantage of the CPA framework to create a new composite analysis that enriches existing CPAs with an additional component that observes the analysis and suggests new precisions to the main analysis when its underlying automaton matches certain situations. As an example we show how specifications can be effectively used as precision mining automata as they inherently carry information crucial for verifying a property.

We also provide a proof-of-concept implementation of our new precision mining technique that makes use of weaving specification automata to generate new precisions. To show that our implementation can severely reduce the amount of refinement iterations needed to verify a program, we provide an experimental evaluation with verification tasks that are created from set of over four thousand Linux kernel modules. In this evaluation we investigate the benefit of our precision mining technique in terms of efficiency and effectiveness and discuss possible problems and how this approach could be further improved. We conclude by summarizing our findings and giving an outlook into future work.

Chapter 2

Background

In this chapter we explain how safety properties of a program can be verified with a reachability analysis, how abstraction is used to limit the cost of this analysis and how counterexample guided abstraction refinement is used to fully automate that process. We also provide background knowledge about symbolic automata and configurable program analysis—two concepts we use for our precision mining technique.

Representation and Reachability Analysis

One way to prove that a program satisfies a certain specification is to show that specific *error locations* defined by that specification cannot be reached during the execution of the program. Therefore a *reachability analysis* constructs an *abstract reachability graph* (ARG)—a representation of the state-space that is explored during program execution.

A Program can be represented as a *control-flow automaton* (CFA) which is a tuple (L, l_0, G) consisting of a set of program locations L with an entry location $l_0 \in L$ and a set of control-flow edges $G \subseteq L \times Ops \times L$. The edges represent the operations of a program like assignments or arithmetic operations. The ARG is then created by iteratively unrolling the CFA of a program and constructing abstract successor states for the current location each time the control flow runs through an edge of the CFA.

A program is then considered *safe* with respect to a specification, if there is no abstract state in the ARG that represents an error location defined by that specification. That means that there is no possible path through the program that passes such an error location.

Predicate Abstraction

For infinite state programs, verification based on reachability analysis may take an unreasonable amount of time or memory or may not terminate at all. With abstract model checking the high cost is traded for precision of the analysis by performing the reachability analysis on a model in an abstract domain [12].

In this context an *abstract domain* $D = (C, \mathcal{E}, [\![\cdot]\!])$ is defined by a set C of concrete states, a semi-lattice $\mathcal{E}$ and a concretization function $[\![\cdot]\!]$. The semi-lattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ consists of a set E of domain elements, the abstract states, with special elements top $\top$ and bottom $\bot$, a preorder $\sqsubseteq \subseteq E \times E$ as well as a total function $\sqcup : E \times E \mapsto E$ called the join operator. The abstract states are mapped to the concrete states they represent by the concretization function $[\![\cdot]\!] : E \mapsto 2^C$. For each abstract state e this mapping gives the set of concrete states e represents.

One abstraction technique that is commonly used in software verification is *predicate abstraction* [10] [2]. Let X be the set of program variables and $\mathcal{P}$ a set of quantifier-free predicates over X, the *predicate abstract domain* consists of boolean formulas over the perdicates of $\mathcal{P}$. A precision π, that consists of predicates from $\mathcal{P}$, guides the abstraction process. During the construction of the ARG, successors of abstract states are created by computing the boolean or cartesian abstraction of the current state and the perdicates from π.

This can be further optimized by a technique called *adjustable block encoding* (ABE) [4]. Program statements are grouped into blocks and abstractions are computed only at the end of each block. One instance of ABE is ABE-Loops where each block contains loop-free parts of a program.

Configurable Program Analysis

Configurable program analysis (CPA) [5] is a framework that allows to combine several analyses into a composite analysis that enables its components to exchange information to strengthen elements of their abstract domain.

Formally a CPA is a tuple $\mathbb{D} = (D, \rightsquigarrow, \texttt{merge}, \texttt{stop}, \texttt{prec}, \texttt{target})$. Its abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$, e.g. the *predicate abstract domain*, defines how the concrete states C from the program are represented. Successors of abstract states are computed via the *transfer relation* $\rightsquigarrow$. The operator `merge` can combine two abstract states into a new one that is an overapproximation of those states, i.e. for two states $e, e^{'} \in E$ it must be true, that $e^{'} \sqsubseteq \texttt{merge}(e, e^{'})$. For a set of abstract states $R \subseteq Z$ and an abstract state e, the termination check $\texttt{stop}(e, R)$ returns *true* if e represents only concrete states that are already covered in R; in other words $[\![e]\!] \subseteq \bigcup_{e' \in R} [\![e^{'}]\!]$ holds true. Otherwise it returns *false*. Dynamic precision adjustment [3] adds the operator `prec` that computes an adjusted precision from an abstract state e, a precision π and a set of reached states R. The last missing component is the operator `target`, which checks if a given abstract state is a target of a reachability analysis, that is, it represents an *error location* defined by the specification.

An essential property of CPAs is that they can be combined into a composite CPA. A new operator $\downarrow$, called the *strengthening operator*, is introduced for the components to be able to exchange information. It is executed at the end of the transfer relation of the composite CPA and allows its components to access information from abstract states of their sibling in order to compute a stronger abstract successor state.

The algorithm executing a CPA (Alg. 1) computes for an initial set of abstract states with precisions W_0 its successors until either a violation is found or the waitlist is empty. The resulting ARG is guaranteed to contain an overapproximation of the set of reachable concrete states [5].

Algorithm 1 CPA algorithm, adopted from [1]

```
Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \texttt{merge}, \texttt{stop}, \texttt{prec}, \texttt{target})$,
    a set $R_0 \subseteq E \times \Pi$ of abstract states with precision,
    a subset $W_0 \subseteq R_0$ of frontier abstract states with precision
Output: a set of reachable abstract states with precision,
    a subset of frontier abstract states with precision
1:  $\texttt{reached} := R_0; \texttt{waitlist} := W_0$
2:  while $\texttt{waitlist} \neq \emptyset$ do
3:      $(e, \pi) := \texttt{choose}(\texttt{waitlist})$
4:      for all $e^{'}$ with $e \rightsquigarrow (e^{'}, \pi$ do
5:          $(\hat{e}, \hat{\pi}) := \texttt{prec}(e^{'}, \pi, \texttt{reached})$
6:          for all $(e^{''}, \pi^{''}) \in \texttt{reached}$ do
7:              $e_{new} := \texttt{merge}(\hat{e}, e^{''}, \hat{\pi})$
8:              if $e_{new} \neq e^{''}$ then
9:                  $\texttt{waitlist} := (\texttt{waitlist} \cup \{(e_{new}, \hat{\pi})\}) \setminus \{(e^{''}, \pi^{''})\}$
10:                 $\texttt{reached} := (\texttt{reached} \cup \{(e_{new}, \hat{\pi})\}) \setminus \{(e^{''}, \pi^{''})\}$
11:         if $\neg\, \texttt{stop}(\hat{e}, \{e \mid (e, \cdot) \in \texttt{reached}\}, \hat{\pi})$ then
12:             $\texttt{waitlist} := (\texttt{waitlist} \cup \{(\hat{e}, \hat{\pi})\})$
13:             $\texttt{reached} := (\texttt{reached} \cup \{(\hat{e}, \hat{\pi})\})$
14:             if $\texttt{target}(\hat{e})$ then
15:                 return $(\texttt{reached}, \texttt{waitlist})$
    return $(\texttt{reached}, \emptyset)$
```

Counterexample Guided Abstraction Refinement

In order to combat the state explosion occurring in model checking, abstraction is the main tool of choice. Indeed there are two problems using this concept alone: First, in the beginning an initial abstract model must be generated, i.e., a decision must be made what information to keep and what to ignore. Second, the level of abstraction needs to find the right compromise between performance and precision. If the abstraction is to coarse, the analysis might find erroneous program paths that do not really exist, but are an artifact of the overapproximation of concrete states. As a result a program might be incorrectly reported as *unsave* due to such a spurious counterexample.

Counterexample guided abstraction refinement (CEGAR) is an algorithm that provides a practical solution to both problems [9]. The algorithm starts with a very coarse abstract model (e.g. everything is abstracted) and automatically refines it iteratively until either a real counterexample is found or the program is reported to be *safe*. The CEGAR algorithm consists of three main parts. At first an initial abstract model is built and model checked. Whenever the model checker reports a counterexample, CEGAR moves to its second step and examines if the counterexample is feasible or not. If the counterexample is feasible, meaning that at least one of the paths represented by the counterexample exists in the real program and ends in an error location, the algorithm terminates and reports the program as *unsafe*. Otherwise the erroneous program path is used in the third step to generate new predicates for the precision and thus, refine the abstract model (e.g. using Craig-interpolation [13]). The new, more precise model then again gets model checked. This loop advances by eliminating at least one spurious counterexample per iteration and continues until the analysis provides a sound result.

Symbolic Automata

As the software verification algorithms we consider commonly deal with symbolic representations of program states and -paths, when using automata, we need to adapt them to cope with such a notation of values. That is what *symbolic finite automata* (SFA) do. In contrast to classical automata theory the alphabet is not given explicitly but implicitly by a Boolean algebra. An easy-to-understand introduction to SFA is given in [15] where the following definitions are largely adopted from.

A symbolic automaton M is a tuple $(Q, \mathcal{A}, q^0, F, \delta)$ where Q is the finite set of *states*, $q^0 \in Q$ is the *initial state* and $F \in Q$ is the set of *final states*. The alphabet $\mathcal{A}$ is an *effective Boolean algebra* $(\mathfrak{D}, \Psi, [\![\cdot]\!], \bot, \top, \vee, \wedge, \neg)$ with a recursive enumerable (r.e.) set of *domain elements* $\mathfrak{D}$, a r.e. set Ψ of *predicates* closed under $\vee$, $\wedge$, $\neg$, $\bot$ and $\top$. The *denotation function* $[\![\cdot]\!] : \Psi \to 2^{\mathfrak{D}}$ is also r.e. and such that $[\![\bot]\!] = \emptyset$ and $[\![\top]\!] = \mathfrak{D}$ as well as for all $\phi, \psi \in \Psi$ holds $[\![\phi \vee \psi]\!] = [\![\phi]\!] \cup [\![\psi]\!]$, $[\![\phi \wedge \psi]\!] = [\![\phi]\!] \cap [\![\psi]\!]$ and $[\![\neg\phi]\!] = \mathfrak{D} \setminus [\![\phi]\!]$. This explains the set of transitions δ of the automaton which is defined as $\delta \subseteq Q \times \Psi \times Q$.

In context of software verification the set of predicates usually consists of the variable names of the verified program. A popular choice of effective Boolean algebra are those who can be implemented via a SMT-solver.

Chapter 3

Pitfalls of Precision Reuse

When optimizing CEGAR-based software verification algorithms one main goal is to reduce the amount of refinement iterations that are needed to rule out all spourious counterexamples. This is because in each iteration an expensive abstraction refinement is performed and parts of the state-space possibly have to be re-explored. In this chapter we revisit precision reuse, an existing approach to avoid refinements by reusing results from previous verification runs. We demonstrate several shortcomings of this technique by performing simple experiments and investigate their causes.

3.1 Precision Reuse

Precision reuse identifies the precisions from past verification runs as valuable intermediate results for other verification runs [8]: We collect the precisions discovered during a verification run and create the union over those. The resulting precision is then saved for future use. Precisions are quite small and therefore can be stored in a simple text-based format [8]. When verifying the same or a similar program one can simply use the previously stored precision for the initial abstract model of this verification task. As a result this model is precise enough to rule out spurious counterexamples that otherwise would cause a refinement step. Thus, the verification of the program needs less iterations to come to a result. If the verified program is exactly the same as when the precision was generated the initial abstract model is precise enough to rule out all spurious counterexamples so there is no refinement needed at all.

The practical benefit of this method shows when it is applied to regression verification. When software is developed there are usually a lot of small changes made to the code resulting in many similar program versions. Therefore, there is a good chance that the abstract models generated during the verification of succeeding program revisions along with their precisions also differ only a little.

An experiment where multiple revisions from different Linux device drivers were verified comparing precision reuse to normal verification runs shows that this technique can indeed significantly reduce the amount of refinements needed as well as the verification time in a regression verification scenario [8].

```
int main() {
 int i=0, a=0;
 int n=20;

 while(i < n) {
  ++i; ++a;
  if(i != a)
   goto ERROR;
 }

 if(i != n)
  goto ERROR;

 return(0);
 ERROR:
 return(-1);
}
```

(a) loopA.c

```
int main() {
 int j=0, a=0;
 int n=20;

 while(j < n) {
  ++j; ++a;
  if(j != a)
   goto ERROR;
 }

 if(j != n)
  goto ERROR;

 return (0);
 ERROR:
 return (-1);
}
```

(b) loopB.c

```
int loop() {
 int i=0, a=0;
 int n=20;

 while(i < n) {
  ++i; ++a;
  if(i != a)
   goto ERROR;
 }

 if(i != n)
  goto ERROR;

 return(0);
 ERROR:
 return(-1);
}

int main() {
 return loop();
}
```

(c) loopC.c

```
int main() {
 int i=0, a=0;
 int n = 20;

 while(i < n) {
  ++i; a+=i;
  if(i > a)
   goto ERROR;
 }

 if(i != n)
  goto ERROR;

 return (0);
 ERROR:
 return (-1);
}
```

(d) loopD.c

Figure 3.1: The Programs used in the experiments. For versions *B*-*D* changes compared to version *A* are marked red

3.2 Investigating Negative Precision Reuse

Although precision reuse can be a great tool for optimizing CEGAR-based algorithms we identified situations where that technique does not work well. There are even cases in which enabling precision reuse has a negative impact on the performance of a verification task, meaning that not enabling precision reuse results in a faster program verification than adding that optimization to the run. Farther we found out that for some simple programs one can intuitively write precisions that outperform those generated during the verification process.

On order to further investigate our observations we designed some experiments that trigger the described situations. In the following sections we first explain how our experiments are set up. After that we analyze the results w.r.t. the impact of precision reuse on the performance and discuss possible reasons of the observed behavior.

3.2.1 Experiment Setup

For our experiments we used odysseus, a fork of the CPAchecker platform[1] [6], as verification tool. All tasks of this experiment are configured to use a standard predicate analysis with ABE-Loops.

Test Programs

We created four different versions of a small C-program, which are shown in Figure 3.1. Each program is designed to demonstrate a specific behavior of precision reuse. Version *A* of the program serves as the base program and the other versions *B*, *C* and *D* are derived from it, each containing one modification compared to the base version.

Version *A* consists of a loop that runs a fixed number of times and increments the loop variable as well as an second variable in each iteration. If those two variables are not equal at the end of each iteration or after the loop, the program enters an error state, otherwise the program terminates ordinarily. In version *B* we changed the name of the loop variable from *i* to *j* altering the syntax of the program but leaving its semantics untouched. For program version *C* the loop is extracted into its own function. The last program version *D* contains changes to the loop body as well as an adjustment of the condition inside the loop such that it again holds true.

Verification Tasks

For each program, versions *A*-*D*, we have six verification tasks with different precisions given as input for precision reuse. The most important of the used precisions are shown in Figure 3.2. They are depicted in a simplified version of the format for precision-files introduced in [8] and consist of blocks that begin with a program location in the form of a function name and a line number followed by predicates that are to be used at that location.

In the following we introduce the tasks of our experiment and explain what purpose they fulfill:

Without Precision Reuse (no_predmap) This is a basic verification run of the program without precision reuse. The precisions generated during this run are saved and used for the next tasks. It also serves as reference for reasoning about the influence of precision reuse on the performance.

Basic Precision Reuse (with_predmap) This is the task with precision reuse added to the verification run. Each program is provided with the precision dumped after the previous task. This task shows how well precision reuse works under ideal conditions.

Precision Reuse With Regression (no_predicate_filter) Each of the program variations is verified using the precision of loopA.c that was generated in the first task (s. Figure 3.2a). All predicates in that precision are kept for the analysis, even if they contain variables that are not present in a specific program version, for example for program *B* most of the predicates contain the old loop variable *i* and thus are obsolete.

[1]https://cpachecker.sosy-lab.org/

```
main l.5:
<= i (+ n -20)
<= i (+ n -19)
...
<= i (+ n -2)
<= i (+ n -1)
<= a (+ n -20)
<= a (+ n -19)
...
<= a (+ n -2)
<= a (+ n -1)
=  i n
<= i a
<= a i
<= (+ n -a) 0
=  (+ n -a -1) 0
=  (+ a -i) 0
<= (+ a -i) 0
<= (+ a -i 1) 0

main l.15:
false
```

(a) loopA.c precision

```
main l.5:
(or (<= i (+ n -20))
   (<= i (+ n -19))
   ...
   (<= i (+ n -2))
   (<= i (+ n -1)))
(or (<= a (+ n -20))
   (<= a (+ n -19))
   ...
   (<= a (+ n -2))
   (<= a (+ n -1)))
=  i n
<= i a
<= a i
<= (+ n -a) 0
=  (+ n -a -1) 0
=  (+ a -i) 0
<= (+ a -i) 0
<= (+ a -i 1) 0

main l.15:
false
```

(b) disjunctive precision

```
main:
= i a
= i n

main l.5:
< i n
= i a
= i n
```

(c) manual precision

Figure 3.2: Precisions used in the experiments

Filtering Obsolete Predicates (with_predicate_filter) This task uses almost the same configuration as the previous one. Only that this time predicates that contain variables that are not present in a program version (and thus are of no use for the abstraction process) are filtered out. Therefore, any effects caused by superfluous predicates should be eliminated.

Optimizing the Precision (manual_precision) Here we have the same configuration as before. But instead of the precision generated in the first task the verifier gets the precision shown in figure 3.2c as input for precision reuse. That precision was written by looking at the input program and intuitively simplifying the precision generated during the first verification task. The expectation is that due to the generalized predicates the loop unrolling can be prevented reducing the cost to verify the programs.

Automating the Optimization Process (or_precision) Again, this task differs from the previous one only by the input precision. This time we tried to construct a precision that resembles the manual one but theoretically could be deduced from the original precision automatically. In this case we simply concatenated the predicates caused by unrolling the loop with `OR`-operations. The resulting precision can be seen in Figure 3.2b. As before we expect a performance gain over reusing the generated precision by avoiding loop unrollings.

	Ref.	Reach	Unroll.	Models	Time Prec.	Time An.	Result
loopA.c							
no_predmaps	22	192	21	25	4.202	5.773	TRUE
with_predmaps	0	192	21	21	1.380	1.748	TRUE
no_predicate_filter	0	192	21	21	1.482	1.853	TRUE
with_predicate_filter	0	192	21	21	1.534	1.762	TRUE
manual_precision	0	21	2	2	0.091	0.152	TRUE
or_precision	0	30	3	3	0.377	0.461	TRUE
loopB.c							
no_predmaps	22	192	21	25	4.519	6.027	TRUE
with_predmaps	0	192	21	21	1.446	1.801	TRUE
no_predicate_filter	22	192	21	463	28.298	29.435	TRUE
with_predicate_filter	22	192	21	21	5.322	6.689	TRUE
manual_precision	22	192	21	25	4.143	5.586	TRUE
or_precision	22	192	21	21	4.778	6.175	TRUE
loopC.c							
no_predmaps	22	236	21	25	4.359	5.845	TRUE
with_predmaps	0	236	21	21	1.390	1.751	TRUE
no_predicate_filter	22	236	21	25	4.405	5.872	TRUE
with_predicate_filter	22	236	21	25	4.925	6.447	TRUE
manual_precision	22	236	21	25	4.174	5.601	TRUE
or_precision	22	236	21	25	4.102	5.453	TRUE
loopD.c							
no_predmaps	9	39	5	20	0.522	0.875	TRUE
with_predmaps	0	39	4	5	0.318	0.400	TRUE
no_predicate_filter	2	192	21	489	9.184	9.622	TRUE
with_predicate_filter	2	192	21	489	9.761	10.256	TRUE
manual_precision	2	30	3	8	0.208	0.464	TRUE
or_precision	2	30	3	16	0.700	0.860	TRUE

Table 3.1: Precision reuse experiment results

3.2.2 Results

Table 3.1 shows the results of the previously described verification tasks. The results are grouped by the program version. The first column shows the number of refinements needed by the CEGAR algorithm followed by the number of abstract states reached during the analysis, the number of performed loop unrollings as well as the number of models generated during abstraction. On the right half of the table the times for the precision adjustment and the total analysis time are given. The results of the verification runs conclude the table.

We now explain the results of our experiments grouped by the verification tasks, as each verification task was designed to show a specific behavior of the verification algorithm in combination with precision reuse.

Without precision reuse (no_predmap) The results of the program versions A-C are very similar as they are semantically (almost) the same. Worth mentioning is that despite the small input program the verification needs a lot of refinements because the loop has to be unrolled by the model checker. The programs are designed to do that, because this huge number of refinements lets us observe the impact of precision reuse more easily as the difference in the results is much larger. The only exception is the run of program D where the different loop body causes the model checker to be able to prove the safety of the program after unrolling only five iterations of the loop.

Basic precision reuse (with_predmap) The second reference task clearly shows that precision reuse works. Each of the runs of this task needed no refinement iteration at all and therefore was significantly faster than its counterpart with the first configuration.

Precision reuse with regression (no_predicate_filter) The expectation of this run is that for program version A the result is exactly the same as in the second task as they are exactly the same. For all other programs only a small part of the reused precision is still valid, and most of the predicates contain variables that are not present at the given location anymore. So the execution time should be about the same as measured in the first task. But this holds true only for program C. Table 3.1 shows an huge increase in the time needed to verify the versions B and D. Not only that precision reuse gives no benefit here, its effect on the performance is extremely negative.

For program B the cause of this is hidden in the abstraction process. When looking at the results one can see that much more models are generated during abstraction than in the other verification runs. This is because additionally to the predicates contained in the initial precision there are those needed for the current program to be verified. A boolean formula is built from those models that is then given to a SAT solver, which in return needs more time for its satisfyability check leading to an overall worse performance.

Program C does not show this effect because the initial precision maps predicates to function names. As most abstractions are computed for the loop that is now in another function, all those useless predicates are not considered most of the time and so do not noticeably affect the performance.

In the case of version D all predicates of the precision are applicable as the variables in program A and D are the same. But those predicates unnecessarily cause the loop of the program to be unrolled completely leading to more refinements and a bigger state-space that gets explored.

Filtering obsolete predicates (with_predicate_filter) A solution for the problem encountered in the previous task is already hinted in [8]. The suggestion is to remove all predicates that contain variables that are not present at the current program location with a quick syntactical check. This is exactly what we do here in order to accommodate for the negative impact of precision reuse we encountered.

Our results confirm that in the case of version B of the program the execution time is again comparable to not enabling precision reuse. The most time consuming part of the verification of the loop programs is the unrolling of the loop. So we cannot expect a faster run than in the first task as the construction of the programs A and B renders that part of the precision generated for version A essentially useless when applied to version B.

But for version D we still observe the same behavior as in the task without predicate filtering. Both program version A and D contain the same variables, they only differ in their loop body. As a result no predicates are filtered out. But the precision of version A causes the verifier to completely unroll the loop of the program even though this would not be necessary as the task without precision reuse showed. Again the effects described before drain the performance of this verification run.

Optimizing the precision (manual_precision) As already explained in the description of the different verification tasks the precision reused here is built by taking an educated guess about what information must be kept in an abstract model of program A such that the verification algorithm needs as few refinements as possible. The precision we came up with (s. 3.2c) showed some interesting results.

First of all we use predicate filtering for this task. So the precision is useless for verifying the programs B and C. The lack of refinement steps for version A shows that using this precision results in an abstract model that is precise enough to rule out all spurious counterexamples. What stands out is that this verification run is way faster than any of the other tasks for version A of our program. Taking a closer look at the results reveals that this time the verifier could prove the program after exploring only three iterations of the loop instead of unrolling it completely, meaning that it is sufficient for the algorithm to compute a smaller part of the abstract model saving a lot of expensive abstraction computations. Saving those loop unrollings is possible because the provided precision contains some kind of loop invariant letting the abstraction process recognize that in the abstract model all loop iterations are covered by the first one.

A similar behavior holds true for the last version D. The precision prevents the loop from being unrolled completely so only a part of the abstract model must be explored by the model checker. For proving the safety of this program only a small amount of additional predicates that differ from the precision must be found. As a result this run is about as fast as reusing the precision generated by this program itself.

Automatize optimization process (or_precision) This last task is there to see if it might be possible to automatically optimize precisions generated during program verification such that the resulting precision performs better than the original one – similar as the manual precision from the previous task does. As described before we did this by comparing the manual (Figure 3.2c) and the original precision (Figure 3.2a) and transforming the later one in a way an algorithm could also do. In the case of our experiment a procedure that optimizes precisions would at first identify similar predicates, here the ones caused by the loop unrolling. Afterwards they get merged by concatenating all similar predicates into a new big one using the Boolean `OR`-operator.

The results show that this precision still outperforms the normal precision of program A by far, although being not as good as the manual one. The effects that cause the improved performance are the same as in the task before only do they occur not as strong.

For the other two program versions there is again no performance improvement, because the precision contains no predicates that are useful for the verification of those. But due to predicate filtering there isn't any negative impact either.

3.3 Conclusion

A detailed experimental study shows that precision reuse is a simple yet powerful possibility to reduce the runtime of program verification tasks [8]. Nevertheless we saw in our experiments that this technique's benefit can vanish when the abstract model generated by the reused precision differs too much from the currently verified program and even turn into *negative precision reuse*. This could easily happen when precision reuse is applied to regression verification where the precision evolves from revision to revision probably outdating more and more parts of the precision. In some situations this effect can be circumvented by cleaning the precision from obsolete predicates before reusing it but in other cases the detection of predicates that harm the performance of the abstraction process is much more difficult.

We also saw that the performance of the verification can be further improved by optimizing the used precision. When looking at the examples of our experiment this seems to relate to the problem of invariant generation. But these experiments also hint that it might be possible to improve precisions by merging predicates automatically in certain situations. A deeper investigation of this idea is out of the scope of this work.

Instead we focus on the concept that lies behind precision reuse: Providing the model checker with additional information that otherwise would only be discovered later during refinement iterations. In the next chapter we introduce a generic method that builds on that idea and makes use of symbolic automata to gather and provide new precisions to the analysis.

Chapter 4

Precision Mining using Symbolic Automata

In the previous chapter we saw that precision reuse can be an effective means to improve the overall performance of CEGAR-based verification algorithms. This happens by providing the abstraction process with additional information such that the generated model is precise enough to rule out spurious counterexamples before they are discovered by the analysis. Nevertheless our experiments showed that there are situations where this technique does not work very well and, in the worst case, can even have a bad impact on the performance. We also found out that the precision generated by the verification algorithm is not necessarily the best one, instead there are precisions that cause a smaller and thus faster to verify abstract model.

In this chapter, we introduce a new technique that builds upon the basic idea behind precision reuse but does not rely on previously computed and stored results. Instead new precisions are generated on-the-fly by using symbolic automata that observe the creation of the ARG and suggest new precisions when they encounter certain situations.

In this chapter, we first define our technique by describing a configurable program analysis that implements this approach. Then we discuss how such a CPA can utilize information available in each verification run through the specification in order to synthesize new precisions and explain how we integrated this technique into odysseus, a verification tool based on the CPAchecker framework.

4.1 A Framework for Precision Mining

Our main goal is to reduce the number of refinements needed to verify a program. To achieve this we must provide the model checker with information about what aspects of a program are important to keep in the abstract model, that is, we must supply new precisions. Of course this has to happen *before* the corresponding parts of the abstract model are created if we want to save any refinements. For these new precisions to be effective, we need to decide (1) what information they contain and (2) at what location they should be used.

From specification automata we know that SFAs can be used to observe and match *locations* in the ARG [1] and when we extend these automata to carry candidate precisions in their state they also yield a way to store *information* new precisions can be drawn from.

The CPA framework allows several analyses to run in parallel and exchange information when needed. We use these properties to add an additional component to existing analyses that runs a precision mining automaton and provides the candidates it generates to the main analysis.

4.1.1 Symbolic Automata for Precision Mining

The automata used to generate new precisions work similar to *specification automata* [1]. They observe the construction and exploration of the programs ARG and suggest new precisions on a match.

Such a *precision mining automaton* is a non-deterministic symbolic automaton $M = (Q, \Sigma, \delta, q^0, F)$ for a given CFA (L, l_0, G) with a finite set of states Q, an initial state $q^0 \in Q$, a set of accepting states $F \subseteq Q$ and a transition relation $\delta \subseteq Q \times \Sigma \times Q$. The elements of the alphabet $\Sigma \subseteq 2^Q \times \Psi_{\mathcal{A}} \times 2^P$ consist of a CFA-edge, a set of assumptions denoted as a predicate of an effective boolean algebra $\mathcal{A}$ with program variables as domain elements and a set of candidate precisions. We also use the notation $q_a \xrightarrow{locations[assumptions]\{candidates\}} q_b$ for transitions.

In order to prevent state-space explosion, *on-the-fly weaving* [1] can be used to encode the automatons operations directly into the transition relation of the analysis. In the same way as specification automata, precision mining automata (1) must not modify variables that were not created by themselves and (2) the conjunction of all guards of all outgoing transitions of each state must evaluate to *true* in order to not affect the completeness or soundness of the analysis [1].

4.1.2 Precision Mining CPA

A CPA to operate those automata can also be deduced from the specification analysis [1]. The *precision mining CPA* $\mathbb{D} = (D, \Pi, \rightsquigarrow, \texttt{merge}, \texttt{stop}, \texttt{prec}, \texttt{target})$ contains an abstract domain D that is constructed from the states of the automaton and a transfer relation $\rightsquigarrow$ that is defined by the abstract domain's transitions. There is currently no use for precisions in this CPA so the set Π of precisions is empty and the precision adjustment `prec` does nothing. The operator `merge` always keeps abstract states separate, i.e., $\texttt{merge}(e, e') = e'$, the operator `stop` checks whether an abstract state is already subsumed by any other state. As there is no special meaning to final states in precision mining automata, the operator `target` can always return *false*.

The most important part of this CPA to our technique is the strengthening operator $\downarrow$ that is called at the end of the transfer relation. Here, precision candidates are generated from the templates the automaton offers after a match. The candidates can be stored in any form that other CPAs can generate a precision of their domain from. For example CFA-edges could serve this purpose.

In existing CPAs that are used for verification, only the strengthening operator has to be extended so that it fetches the candidate precision offered by precision mining CPAs. Later in the precision adjustment operator of the analysis the candidates can be used to improve the current precision.

4.1.3 Templates as Candidate Precisions

Our precision mining technique can be made more flexible by using templates instead of predicates. A *template* is a formula over unknown variables that take values over some subset of a given set of predicates [14]. In this case, the unknown variables can be substituted by variables occurring in the description of the CFA-edges. This has the advantage that the automaton adapts to new situations. For example, when the name of a variable that occurs in a candidate changes (e.g. with a new revision of the program), the automaton needs not to be modified because the template takes care of substituting the correct variable name. Another, probably more important, use case is to use templates to make a precision mining automaton match classes of situations instead of individual ones. Consider a precision mining automaton that wants for each call of a function `foo(a)` to provide a precision of the form $\texttt{a} \neq \texttt{0}$. With templates, the automaton needs only one state q with a single transition $q \xrightarrow{foo(\$1)[]\{\$1 \neq 0\}} q$. When the verification algorithm runs through a CFA-edge that calls the function `foo(a)`, the precision mining CPA takes care of substituting the template variable \$1 with the real variable of the CFA-edge and providing the instantiated candidate precision to the analysis. It is also possible to allow the assumptions to be templates to further control how candidate precisions are generated.

4.2 Using Specifications for Precision Mining

When using pure weaving specification automata [1], it is a good idea to look at assumptions that they introduce to the code. This is because those assumptions typically encode information that distinguishes between correct and erroneous program locations. For the abstract model to be able to separate these locations into different abstract states and therefore, rule out spurious counterexamples – the precision of the program analysis must contain that information.

In the simplest case one can derive a precision mining automaton from a specification automaton that has only one state and one transition for each assumption the specification makes that presents this exact assumption as a candidate precision. Assume a specification that manages its internal state in a variable `state` and enters an error location whenever the function `foo()` is called and the assumption $\texttt{state} == \texttt{0}$ is true. The precision mining automaton generated from this specification has a transition $q \xrightarrow{foo()[state==0]\{state==0\}} q$ with $q \in Q$ being the automatons only state.

Without any information about the variable `state` the analysis would report a counterexample that is spurious when it first reaches a call to the function `foo()` as the computed abstract state would include a concrete state where `foo()` is called and $\texttt{state} == \texttt{0}$. In order to eliminate this counterexample a refinement iteration would be necessary. But if we mine the specification for precisions with this method, the analysis is told to separate all states where $\texttt{state} == \texttt{0}$ into a different abstract state.

We presented templates as a way to make precision mining automata more flexible. And they can also be used in matcher descriptions and assumptions of specification automata [1]. An example for this is a specification that ensures

```
int main() {
  int condition = nondet();

  if (condition)
    lock();
  if (condition)
    unlock();
}
```

Figure 4.1: The program *lock.c* used to demonstrate precision mining

that no division by zero occurs in the program. The precision mining automaton generated from such a specification has again only one state q and a transition $q \xrightarrow{\$?/\$1[]\{\$1 \neq 0\}} q$, where \$? and \$1 are template variables. When the template variables get replaced by actual program variables (or constants) during the analysis from a statement a/b, our precision mining automaton generates the candidate precision $b \neq 0$.

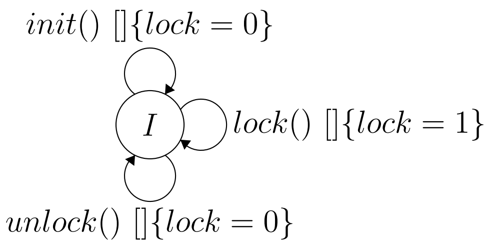
During the runtime of the analysis we can take advantage of additional information that is not statically available. In particular we can track the values of specific variables, either in a map or with an explicit value analysis [7], and use those values to generate more specific precisions. For example this could be used to reduce the amount of loop unrollings by approximating loop-invariants with values from few loop iterations. The use of CPA makes this theoretically easy to implement although we do not focus on that idea in this work.

4.2.1 Example

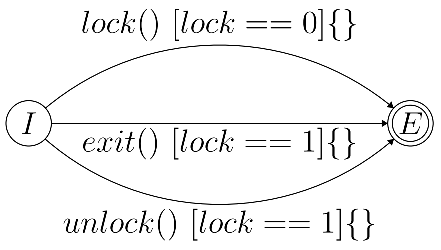
To gain a better understanding of how the precision mining automata and CPA work we now demonstrate our precision mining technique using a specification automaton and an example program.

Program We verify the C-program shown in Figure 4.1. The program first defines a condition by assigning a random value to the variable `condition`. If `condition` $\neq 0$ holds true two functions `lock`() and `unlock`() are called.

Specification The specification used for verifying this program checks for a correct usage of the functions `lock`() and `unlock`(). As expected, those functions implement a *lock*: the function `lock`() acquires the lock and `unlock`() releases it. This mechanism is used correctly iff. (1) the lock is never acquired twice, (2) `unlock`() is only called when the lock is held and (3) the lock is released when the program terminates. Figure 4.2 shows this specification as automata. We write transitions in the same way as with precision mining automata but instead of candidate precisions the curly braces now contain code that gets weaved into the ARG. The specification is split into two components: The first part (Figure 4.2a) is the environment that instruments the code with additional information that is needed by the specification. This automaton has


$init()$ $[]\{lock = 0\}$
I
$lock()$ $[]\{lock = 1\}$
$unlock()$ $[]\{lock = 0\}$

(a) *environment* automaton


$lock()$ $[lock == 0]\{\}$
I
E
$exit()$ $[lock == 1]\{\}$
$unlock()$ $[lock == 1]\{\}$

(b) *observer* automaton

Figure 4.2: The *lock*-specification divided into *environment* and *observer* part

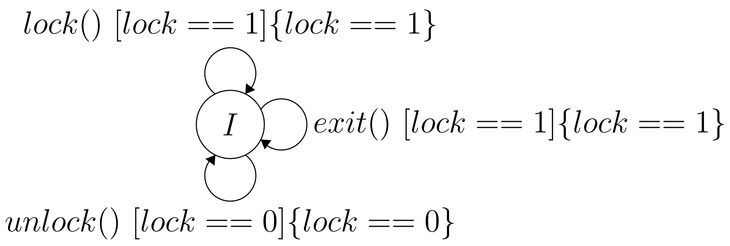

$lock()$ $[lock == 1]\{lock == 1\}$
I
$exit()$ $[lock == 1]\{lock == 1\}$
$unlock()$ $[lock == 0]\{lock == 0\}$

Figure 4.3: *Precision mining* automaton derived from the lock-specification

only one state and several transitions that match a set of *CFA-edges* (*init*() matches the begin of the program and *exit*() anywhere it terminates) and encode a set of*operations* into the CFA. For example, if the analysis passes a call to the function `lock()` the environment automaton sets the value of the variable `lock` to 1. The second part (Figure 4.2b) observes the ARG and jumps to an error state when it encounters a violation of the specification. To detect violations it relies on the state provided by the environment automaton.

Execution To enable precision mining we first need an automaton that fulfills this purpose. We can create such an automaton by applying the procedure described in the previous section to our specification. The resulting automaton can be seen in Figure 4.3.

When the analysis starts with an initially empty precision the first thing that happens is that the environment automaton inserts a new variable `lock` and sets its value to 0. After `condition` is initialized the reachability analysis comes to the first if-construct. Here the ARG splits into two paths because the condition could either be *true* or *false*. On the path where the condition was true we would now find the first counterexample. As the abstraction does not take the variable `lock` into account, the analysis finds a program path where `lock` == 1 when `lock()` is called. This is where the precision mining automaton comes into play. It matches the function call and suggests the new precision `lock` == 1. Thus, the abstraction for this location computes an abstract state for the case `lock` == 1 and one for !(`lock` == 1). The reachability analysis is then able to mark the path where `lock()` is called while `lock` == 1 as not reachable. The same thing happens later for the call to the function `unlock()` and in the end right before the program terminates.

In all three cases our precision mining automaton provides the analysis with exactly the information it needs to not produce spurious counterexamples saving all three refinements the verification would have needed otherwise.

4.3 Implementation

Extracting new precisions from the specification as described in the previous section is a very easy and rewarding application of our precision mining technique. To be able to evaluate this method we integrated it into odysseus, a fork of the model checking tool CPAchecker [6].

For convenience and to better fit into the tool, our implementation differs a bit from the formalism we describe in this chapter. The specification analysis [1] already provides all the information we need for precision mining. Therefore, instead of writing a new CPA we extended the existing CPA that handles the specifications with a new feature, that enables it to generate candidate precisions from specifications.

4.3.1 More on Specification Automata

As seen in the example, the specifications we use are purely weaving and are split into two parts. The first part, the *environment automaton*, is responsible for adjusting the state encoded by the specification; the second part, the *observer automaton*, represents the actual specification, i.e., it matches situations that violate the specification and reports them to the analysis. Both automata have only one state and encode all information they need directly into the CFA of the program being analyzed. Besides preventing state-space explosion caused by the automaton CPA, this format makes specifications easier to comprehend and helps extracting candidate precisions from the specification as candidates are only taken from the observer part.

4.3.2 Extention of Existing Analyses

Very few changes are needed for the new feature. During the strengthening of the specification's transfer-relation we query the current state for assumptions made by the observer automaton that are associated with a possible match. If such assumptions exist they are transformed into an intermediate format and stored as part of the (specification-CPA-)state. We choose CFA-edges to represent candidate precisions because they are easy to handle and mostly analysis-agnostic in CPAchecker.

The analysis used for verification then must fetch the candidates from the abstract state of the specification's CPA, transform the CFA-edges into precisions of its domain, and merge the newly generated precisions with its current one. In CPAchecker the precision-adjustment operator of the used analysis also has access to the state of other CPA's so we implement this second step there instead of during the strengthening operator. We choose CPAchecker's predicate abstraction for our implementation. For this analysis, at the begin of the precision adjustment operator, the candidates are transformed into predicates which are then added to the current precision. If an abstraction is computed in this iteration of the CPA algorithm, the precision with the newly added predicates can already be used.

Being a popular choice for model checking we focus on compatibility with predicate abstraction, but any kind of analysis can be extended to work with our implementation as long as it can construct precisions from CFA-edges.

Chapter 5

Experimental Evaluation

In order to to investigate the potential of our precision mining technique and its impact on the efficiency and effectiveness of program verification we perform an experimental evaluation. We verify programs from a set of several thousand Linux kernel modules using predicate analysis and compare the performance of our implementation of precision mining from specifications (s. Chapter 4) to a configuration that does not make use of additional precisions.

In this chapter we first present the research questions that guide this experimental evaluation, then we describe the experiment setup with the used programs, specifications and execution environment. Afterwards, we answer our research questions with the gathered results and discuss advantages and disadvantages of precision mining that are revealed by the experiments as well as threats to the validity of our evaluation.

5.1 Research Questions

The goals of this evaluation are formulated in four research questions that are separated into two groups addressing the efficiency and effectiveness of our new technique.

Efficiency of Precision Mining First we want to know if and how precision mining from specifications affects the performance of the analysis. In our investigations we focus on how many refinements can be saved with this method over a normal predicate analysis and what speedup can be achieved.

RQ 1.1. *How many verification tasks need less refinements to be verified and how many refinements can be saved by enabling precision mining compared to tasks not using this technique?*

RQ 1.2. *How many verification tasks can be solved more efficiently in terms of CPU-time of the analysis and what is the speedup gained by enabling precision mining compared to tasks not using this technique?*

RQ 1.3. *How does the ratio of saved refinements affect the speedup of the analysis and are there other factors affecting the performance?*

Effectiveness of Precision Mining The saved refinements and performance gain might increase the effectiveness of the verification tool, i.e., more programs are verifiable within the given resources when using precision mining from specifications than without.

RQ 2.1. *How many more verification tasks can be completed within the given resources by enabling precision mining compared to tasks not using this technique and are any results lost?*

5.2 Setup

Our experiments are based on the replication package presented in [1]. We use the set of 4336 Linux kernel modules as well as six of the specifications that describe a correct usage of the Linux kernel API to build our verification tasks. Our implementation is contained in the branch *precision_automaton* of the CPAchecker fork odysseus. For these experiments we used the version labeled with the tag *prec_mining*.

Benchmark Suite As stated earlier, the programs we use for our evaluation are taken from the replication package shipped with the paper about multi-property verification [1]. The package contains 4336 modules of the Linux kernel version 4.0-rc1 that are prepared with the LDV toolkit[1], which enriches the modules with an environment model of the Linux kernel and adds entry-functions to the code.

The properties we use for the verification runs describe the correct usage of a part of the Linux kernel API and are given as six specification automata. The specifications are also taken from the earlier mentioned replication package, although their representation is changed to be pure weaving and they are split into environment and observer as shown in the last chapter. Table 5.1 shows a description of these properties.

Not all of the specifications are relevant for each program of the test set so we consider only such combinations where the specification is relevant to the program. Additionally we filter out all tasks where the analysis fails due to internal errors of the verification tool. Applying these restrictions leaves us with a total of 2038 combinations of programs and specifications. How many programs per specification remain can also be seen in Table 5.1.

Experiments Our experiments use two different configurations of the verification tool. The first one uses CPAchecker's *predicate analysis* with adjustable block encoding to verify each of the 2038 specification-program combinations from our test set. These tasks form the baseline of our experiments against which efficiency and effectiveness of other tasks are measured. The second configuration makes use of our implementation of *precision mining* as described in section 4.3 of the previous chapter. Candidate precisions are generated from the specification and prorosed to the predicate analysis CPA on-the-fly. This is the only change in the analysis compared to the baseline configuration. Together, this results in 4076 different runs of the verification tool.

[1]http://linuxtesting.org/project/ldv

Property	Description	# Progs
LDV_08_1a	Each module that was referenced with `module_get` must be released by `module_put`.	116
LDV_32_1a	The same mutex must not be acquired or released twice in the same process.	812
LDV_43_1a	Each memory allocation must use the flag `GFP_ATOMIC` if a spinlock is held.	735
LDV_68_1a	All resources that were allocated with `usb_alloc_urb` must be released with `usb_free_urb`.	115
LDV_129_1a	An offset argument of a `find_bit` function must not be greater than the size of the corresponding array.	117
LDV_134_1a	The probe functions must return a non-zero value in case of a failed call to `register_netdev` or `usb_register`.	143

Table 5.1: Used specifications (adapted from [1]). For each specification, its name, a description and the number of programs it is relevant for are given

Our experiments were performed on machines that have a Quad Core CPU installed running at 3.4 GHz (Intel Core i7-2600) and are equipped with 32GB of RAM. Each task has a run-time limit of 15 minutes of CPU time and is restricted to 15GB of RAM; the JVM heap size is limited to 13GB. Those limitations to resources are controlled with BenchEexc[2], a freely available benchmarking tool. In order to cope with variations in measurements caused by the just-in-time compiler of the JVM we force it to compile most of the bytecode on startup.

5.3 Results

For 1487 of the 2038 program-specification combinations both tasks, the one with, and the one without precision mining, can be solved within the given resources of which 1227 times the verifier proves the program safe and 260 times a violation is found; for 515 combinations, at least one of the tasks runs into a timeout. Cases where either the baseline task or the precision mining task exceeded the resource restrictions are excluded from the discussion for most research questions as they would distort the result. Instead, they are considered more closely in the answer to *RQ 2.1*.

Detailed information for the fifty best, median and worst results regarding speedup can be seen in Appendix A. For each listed run it shows the *number of refinements* occurred, the used *CPU time for the analysis*, the *fraction of refinements* the task with precision mining needed compared to the corresponding task without this technique and the *speedup* precision mining achieved as well as the result of the verification (i.e. if a violation was found or a timeout occurred). When not stated otherwise, time is given as seconds of CPU time and is rounded to two significant digits.

[2]https://github.com/sosy-lab/benchexec

	True			**False**		
	% Ref.	Speedup	Count	% Ref.	Speedup	Count
LDV_08_1a	0.52	1.02	6	0.89	0.99	85
LDV_32_1a	0.29	1.3	551	0.61	1.1	94
LDV_43_1a	0.61	1.15	494			0
LDV_68_1a	0.94	0.98	18	0.94	1.26	79
LDV_129_1a	0.0	1.43	95			0
LDV_134_1a	1.11	1.01	63	1.0	0.97	2
All runs	0.45	1.23	1227	0.8	1.11	260

Table 5.2: Evaluation results grouped by specification and type of result; all values are arithmetic means

	No Timeout			**Timeout**		
	% Ref.	Speedup	Count	% Ref.	Speedup	Count
LDV_08_1a	0.86	0.99	91	0.89	1.0	25
LDV_32_1a	0.34	1.27	645	0.52	1.3	167
LDV_43_1a	0.61	1.15	494	0.73	1.2	241
LDV_68_1a	0.94	1.21	97	0.98	3.56	18
LDV_129_1a	0.0	1.43	95	0.0	1.15	22
LDV_134_1a	1.11	1.01	65	0.98	1.0	78
All runs	0.51	1.21	1487	0.69	1.27	551

Table 5.3: Evaluation results grouped by specification and whether a timeout occurred; all values are arithmetic means

For our explanations we distinguish between cases where the result of the analysis is *true* and those cases where the result is *false*. We also separate all tasks where the analysis ran into a timeout because an incomplete analysis delivers distorted results. This group also contains cases where for one of the configurations a timeout occurs but for the other one a result can be given.

As the verification costs differ heavily between the programs of our test set (some programs take seconds to be verified, other tasks cannot even finish in the given time), we need measures that are agnostic to this fact in order to compare the results for different programs. That is, we use the *ratio* of the sizes of interest between the baseline tasks and those incorporating our precision mining technique. More precisely, we focus on *how many refinements* the latter of these configurations needs in comparison to the first one as well as the *speedup* that can be achieved in this way. The ratio of refinements is the main size of interest as reducing the number of refinements is the primary goal of our approach. Thereby we expect to increase the efficiency of the verification algorithm in terms of analysis time, hence the gained speedup is of interest. To make this size more meaningful, it only refers to the time needed for the analysis, meaning, that all preprocessing steps are omitted. This does not hide any side effects caused by precision mining because our implementation does not require any preprocessing.

Tables 5.2 and 5.3 contain a summary of all results of our experiments grouped by specifications and verification result or whether a timeout occurred or not. In each case, the average fraction of how many refinements compared to the baseline experiments the tasks with precision mining needed, the average speedup and the number of tasks contained in each group are given.

5.3.1 RQ 1.1: Efficiency in Terms of Refinements Saved

On average, the tasks that use precision mining need only 56% of the refinements the tasks without this technique do. When looking only at those tasks, for which the analysis can actually deliver a result (i.e. they do not run into a timeout) this number improves to 51% meaning that in this experiment precision mining is able to cut the number of refinements in half. If we just consider runs where the program does not violate the specification, only 45% of the refinements from the tasks without precision mining are needed. For violating tasks, 80% of the refinements are needed on average – a worse result than for non-violating tasks. This behavior is expected because for finding counterexamples it is often necessary to further unroll a specific path through the CFA of the program so there are more chances that refinements are needed that precisions synthesized from the specification cannot prevent (e.g. when the control flow diverges at loops or conditions). Additionally only a part of the abstract model is explored and thus, potential for saving refinements is lost.

However it depends heavily on the specification how efficient in terms of saved refinements our technique is. On the one end, all refinements can be saved for the specification LDV_129_1a. These tasks probably work so well, because this specification just compares two parameters of a function call and therefore, it affects only single locations of the program in isolation (one per occurrence of the function call). No information must be tracked over a longer path through the ARG and all the information needed to handle this situation can be delivered via precision mining from the specification. This specification also demonstrates the benefit of templates: the specification automaton matches the relevant function call with the two parameters as template variables. The instantiated version of the template is used for the violation-check of the specification as well as for the generation of the candidate precision. On the other end we observe an increase in refinements needed for the specification LDV_134_1a. Indeed this bad result is dominated by two tasks with an overall small number of refinements (1 or 2 refinements respectively) where the corresponding run with precision mining uses six to seven times more refinements. If we exclude these two cases, we observe a similar result to the specifications LDV_08_1a and LDV_68_1a, where only a small fraction of the refinements can be saved, although in these cases the bad increase in efficiency can be explained by the large number of violations reported for these specifications. The box plots in Figure 5.1 as well as Figure 5.2 further clarify how differently precision mining behaves for different specifications: Not only do they vary in overall efficiency, but also in the distribution of their results. For some Specifications, the results are tightly packed together, while for other specifications they are scattered over a wide range of values.

This makes it hard to predict the performance of a given verification run when using this technique and aggravates reasoning about the general efficiency

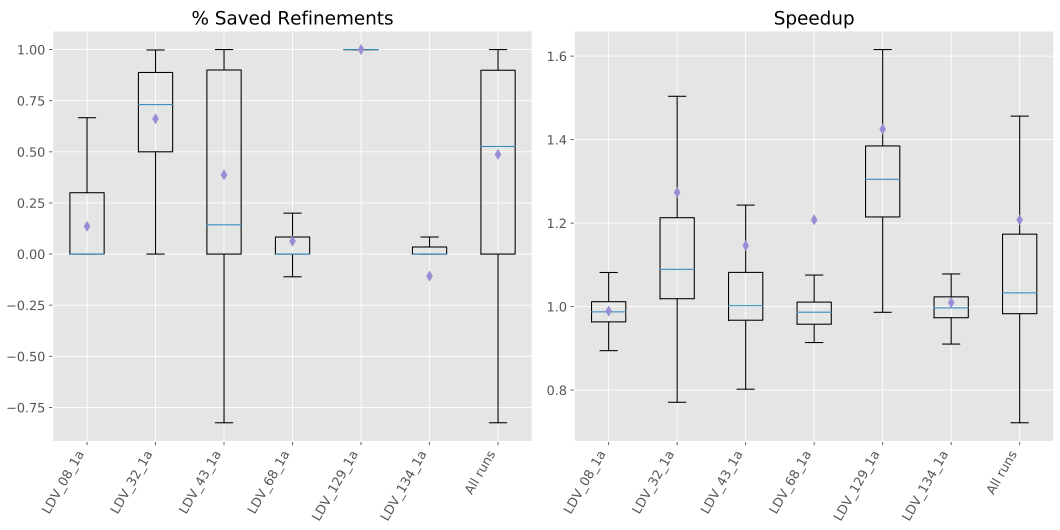


Figure 5.1: Box plots of the percentage of saved refinements (left) and the speedup of the analysis (right) for each specification and for all results together. Only results with no timeouts are considered and outliers are not shown.

of precision mining from specifications. Nevertheless our experiments show that our approach can severely reduce the amount of refinements in many cases.

5.3.2 RQ 1.2: Efficiency in Terms of Analysis Speedup

Despite the good efficiency when looking at the number of refinements the analysis time shows only a moderate speedup. On average the tasks with precision mining are 1.2 times faster than the ones without. Tasks without a violation of the specification show a slightly higher speedup than tasks that do violate the specification (speedup of 1.2 vs. 1.1). The “speedup” for the tasks with timeouts origins in cases where only one of the configurations can provide a result, and therefore, it has no meaningfulness.

Our technique itself introduces almost no noticeable overhead to the analysis. Each candidate precision must be first converted into a CFA-edge and then into a predicate, so the costs for this procedure increase linearly with the number of candidates. However, each newly added predicate makes the formulas used in the abstraction procedure larger and thus, more expensive to solve for an SMT-solver. The additional time spent this way during abstraction computations counterweights the speedup gained by saving refinements negatively affecting the total speedup. Again, the specification has a huge influence on the measured efficiency. One more time, the specification LDV_129_1a gets the best results and LDV_08_1a and LDV_134_1a with their huge amount of detected violations the worst showing no speedup on average.

The lessons learned from analyzing the efficiency in terms of analysis time are similar to those of the first research question: Both, the program and the specification largely influence the result and the measurements are scattered over a more or less wide range. Only a coarse tendency of the outcome can be

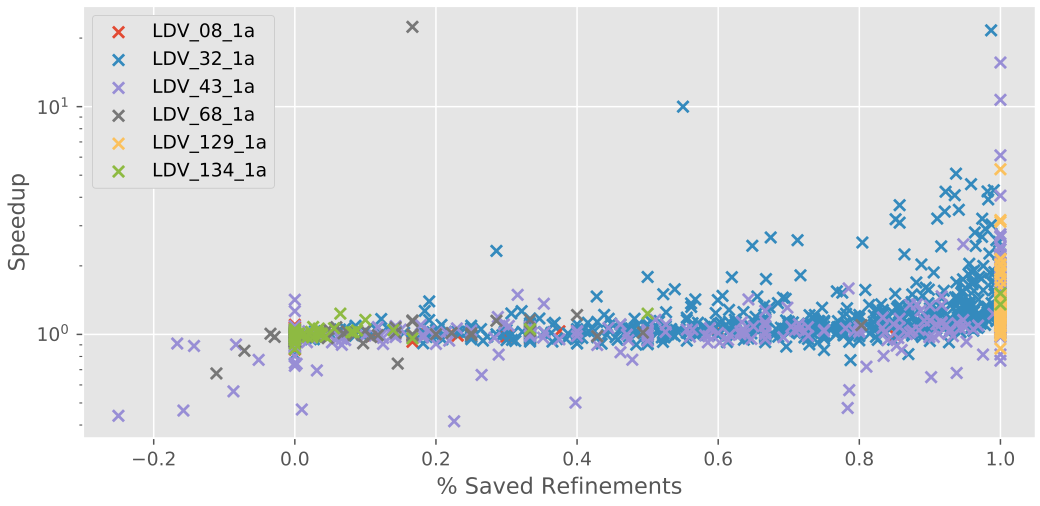


Figure 5.2: Relationship between saved refinements (x-axis) and analysis speedup (y-axis; log scale) grouped by specification. For a better representation two outliers in negative x-direction are omitted.

given by analyzing the results of many different verification tasks and according to that, precision mining as in our implementation yields only a moderate increase in efficiency.

5.3.3 RQ 1.3: Saved Refinements and Analysis Speedup

We want to know, how the saving of refinements via precision mining from specifications is reflected in the achieved speedup and what other other factors there are which influence the speedup.

Figure 5.2 shows a scatter plot of our results that reveals the relationship between saved refinements and speedup. The x-axis is labeled with the percentage of refinements saved when using precision mining over not using this technique. A value of one means that all refinements could be saved, the value zero expresses that both runs needed the same amount of refinements, and values below zero suggest an increase in refinements when using precision mining. As tasks with timeouts carry misleading results, we excluded them for this plot. For a better visual representation two outliers with an huge increase in the amount of refinements (the same two tasks mentioned in RQ 1.1) are also not shown. For most of the tasks that show an decrease of refinements of 60% or less the speedup varies around one, with several outliers in both, positive and negative direction. It is approximately at a level of $60 - 70\%$ of saved refinements where the time saved with the refinements begins to dominate other costs that come with additional predicates and the speedup tends to get bigger. When nearly all refinements can be saved the achieved speedup grows rapidly in many cases, although other tasks still manage to have a worse performance than their counterparts using the baseline configuration.

In the answer to the previous research question we saw, that precision mining increases the cost for the abstraction because the additional predicates add to the complexity of formulas that must be solved by a SMT-solver. For precision reuse to be efficient in terms of analysis time, the decrease in refinements caused by the additional predicates must outperform the overhead they introduce to the abstraction computation. The overall picture given by our experiments suggests, that the reduction of refinements must be severe (here: $> 60\%$) in order to really gain a noticeable performance increase, and even then the actual benefit depends heavily on the verified program. Otherwise the computational overhead introduced with additional predicates cancels out any savings gained with the smaller number of refinements.

5.3.4 RQ 2.1: Effectiveness of Precision Mining

When using our precision mining technique the analysis can provide additional results in 25 cases where it otherwise runs into a timeout. In one of these tasks the time limit of $900s$ is only barely missed so this result might be an artifact caused by a variation in the execution of the JVM. But the other 24 of those tasks can be solved easily within the given resource limits with the fastest runs finishing in under half a minute. In most of these cases the crucial point that makes these programs verifiable is indeed the time we saved by needing less refinements thanks to the information provided by the synthesized precisions. This is true especially for tasks, where the time spent for refinements takes a substantial fraction of the whole analysis time. In other cases the additional predicates in combination with the saved refinements enables other parts of the analysis, like the precision adjustment (including the abstraction computations) or the transfer relation, to be cheaper.

In contrast, we also lose 11 results when enabling precision mining. However this time 4 of the results are unclear because the tasks without precision mining miss the time limit only by a few seconds. The reasons of the lost results vary from an explosion in the cost of the refinement steps to a wider unrolling of the CFA resulting in an overall higher computation time. Interestingly, almost all of the tasks with lost results use the specification LDV_43_1a, although, the exact reason for this specification loosing that much results could not be identified with these experiments. A deeper investigation of this matter and development of possible countermeasures is out of scope of this work and left as subject for future work.

Summary, the use of precision mining from specifications gained 24 results while loosing only 7. So we can say that this technique improves the overall effectiveness of the analysis used for this experiment.

5.4 Discussion

Our results show that precision mining in many cases can severely reduce the number of refinements needed to verify the program and a lot of programs can be verified without any refinements at all. We also achieve some additional results that are not accessible with a normal predicate analysis. However we hardly gain a noticeable speedup for most tasks. As a cause we identified an increase in the complexity of the abstraction computations introduced by the additional precisions that mitigates the speedup gained with saving refinements.

We found out that the specification plays a key role in how effective this technique is, although to this point we do not know what part of a specification in particular is beneficial or detrimental to the efficiency of precision mining. The good results achieved with the specification LDV_129_1a (i.e. none of the tasks needs any refinements) suggest that specifications with a very local scope in the program might perform better than specifications that have to track internal state over long paths through the program. A deeper understanding of such effects could aid the improvement of this technique.

If we provide useless or bad information the wrong parts of the abstract model get refined resulting in a different, potentially larger state space to explore. Therefore the choice of candidate precisions is crucial to the success of precision mining. Our implementation is based on the presumption that all assumptions made by the observer part of the specification automaton is helpful for refining the abstract model in the right way, but duplicates or superfluous precisions might be generated. CPAchecker eliminates duplicates and also filters out some unwanted predicates mitigating possible negative effects arising from those. The high rate of saved refinements suggests that most of the information we supply via precision mining is indeed useful. Therefore, we find our choice of candidates to be a good compromise between ease of implementation and resulting efficiency – at least in terms of saved refinements.

The bad performance gain however leaves room for improvement. Right now, in most cases the overhead introduced to the analysis by the additional precisions is too large for our implementation to be efficient when talking about computation time. A different technique for deriving precision mining automata from specifications could improve this technique either by a different structure of the automaton or by a more careful selection of candidate precisions.

5.5 Threats to Validity

In this section we discuss characteristics of our experiments that might put a threat to their internal or external validity.

5.5.1 Internal Validity

First of all, we saw that the used specification has a huge impact on the effect of our technique. We chose specifications that represent different concepts (e.g. one specification describes a mutex, another one checks for correct function parameters, etc.) in order to gain a more general picture of the effects of precision mining. Nevertheless we can take only a few specifications into account and they stem all from the same application domain perhaps threatening the internal, as well as external validity of this evaluation.

Even for the same specification, the results of different programs are very dissimilar. We ensure this to not endanger the internal validity of our experiments with the large size of our test set.

The resource limits chosen for our experiments are based on values from other experiments that use the same input and verification tool, thus, most of the tasks can be solved within these limits. When a timeout occurs or a violation is found in a task the abstract model is not built completely and the results are missing refinements. Such cases must therefore be looked at separately from successful runs in order to not affect internal validity.

5.5.2 External Validity

The selection of the used programs might be the biggest threat to the external validity in these experiments. Instead of artificially creating test data we rely on real-world software and specifications in order to gain credible results. Device driver and kernel modules are an important application domain for software verification and the size and diversity of our test set ensures external validity at least in this domain. However our test set is not necessarily representative for all kinds of software.

Different verification tools or abstract domains might react differently to the precisions we synthesize from the specifications. In this experiment we focus on CPAchecker's predicate analysis as the predicate abstract domain is widely used, especially with CEGAR based verification. But other tools or the use of other abstract domains may yield results differing from ours.

Chapter 6

Conclusion

Software verification is a computational hard problem and abstraction and CEGAR automate this process and make it accessible for many use cases. One approach to improve the performance of CEGAR-based verification algorithms is to short-circuit the execution by providing the analysis with information such that it needs less refinements until the abstract model has reached the right precision to rule out all spurious counterexamples.

An existing technique that accomplishes this is precision reuse and it is known that it has limitations [8]. Despite being designed for regression verification we identified situations where small changes to the source code – changes that could occur during software development – can cancel out the performance improvements gained by the use of this technique. Admittedly our examples are artificially created in order to investigate such situations that expose the shortcomings of precision reuse. But we identify that the main problem in all investigated situations remains the static nature of the reused information so this technique can not adapt well to new verification problems. Another obvious shortcoming of precision reuse is that it cannot be used for a single verification run of a program as the information being reused must at first be generated.

Therefore, we introduce a new technique to find new precisions that does not rely on static information gathered during a past verification run but generates new precisions *on-the-fly*. This process is guided by symbolic automata that use templates in order to *adapt* the candidate precisions they generate to the program under investigation. In that way we can reduce the amount of refinements the analysis needs by refining the precision that guides the abstraction process with the generated candidates on-the-fly. Specifications in the form of automata provide a valuable source to extract candidate precisions from, because they typically contain information that distinguishes between correct and erroneous program paths. This is exactly the information needed for the abstract model to be able to avoid spurious counterexample paths and additionaly, specification automata are easy to transform into precision mining automata. The CPA framework makes our technique easy to integrate into existing analyses by adding an additional CPA that handles the precision mining automata. Only a minor addition to the analysis' CPA must be made such that candidate precisions can be fetched from the new CPA.

We confirm that our approach increases the effectiveness and efficiency of a predicate analysis by integrating it into CPAchecker and performing an experimental evaluation with Linux kernel modules involving over four thousand single verification tasks. The results show that generating precisions from specifications can reduce the amount of refinements significantly and does not introduce a noticeable overhead by itself. The verifier is also able to solve more problems than it was before. Although in most cases the performance in terms of runtime increases only moderately because the additional predicates make the abstractions more expensive reducing the overall performance gain. To mitigate these negative effects is the main challenge in order to improve this technique's performance.

Future Work For future work it would be interesting to apply our technique to other abstract domains, for example *explicit value analysis* [7]. Our implementation and experiments make only use of specifications as precision mining automata, but there are other situations where our technique could improve the analysis, e.g., by using conditions of loops or branches as candidate precisions in certain cases. We also suggest to include runtime information in the mining process, e.g., by tracking the values of certain variables in a map or via explicit value analysis and use these values to synthesize new precisions. In that way candidates can be produced, that are more specific to the exact situation and thus, perhaps more rewarding.

Appendix A

Experimental Study Relults

The following tables show detailed results for some tasks from the experimental evaluation in Chapter 5. The first table contains the fifty results with the best speedup, the tasks in the second one form the median and the last table presents the fifty results with the worst speedup. Program-specification combinations, where for both tasks a timeout occurred are excluded as their results are meaningless. The name of the verified program and the used specification, the CPU time of the analysis excluding preprocessing time, the number of needed refinements and the verification result are given for each task. Additionally the tables contain the ratio of refinements needed when using precision mining over normal predicate analysis and the achieved speedup.

It can be seen, that saving refinements does not guarantee speedup and, in the other way round, sometimes not saving many refinements can still result in a huge speedup.

Table A.1: Top results

Program	Spec.	An. CPU time		# Refinements		Result		Ref. used	Speedup
		no	yes	no	yes	no	yes		
drivers-bluetooth-bcm203x.c	68_1a	880.15	18.6	5	5	UNKN	FALSE	1.0	47.32
drivers-mfd-dln2.c	43_1a	880.18	21.91	176	20	UNKN	TRUE	0.11	40.17
drivers-hwmon-w83793.c	32_1a	879.7	30.3	2430	32	UNKN	TRUE	0.01	29.03
drivers-hid-usbhid-usbmouse.c	68_1a	447.26	19.96	6	5	FALSE	FALSE	0.83	22.41
drivers-auxdisplay-cfag12864b.c	32_1a	235.52	10.89	227	3	TRUE	TRUE	0.01	21.63
drivers-net-wireless-rtl818x-rtl8180-rtl818x_pci.c	43_1a	456.76	29.24	231	0	TRUE	TRUE	0.0	15.62
drivers-usb-host-r8a66597-hcd.c	43_1a	745.65	69.61	235	0	TRUE	TRUE	0.0	10.71
drivers-video-fbdev-broadsheetfb.c	32_1a	342.09	34.21	409	184	TRUE	TRUE	0.45	10.0
drivers-net-ethernet-smsc-smc91c92_cs.c	43_1a	74.94	12.26	36	0	TRUE	TRUE	0.0	6.11
drivers-power-bq27x00_battery.c	32_1a	881.02	163.24	314	1	UNKN	TRUE	0.0	5.4
drivers-net-ethernet-sis-sis190.c	43_1a	879.07	163.74	564	110	UNKN	TRUE	0.2	5.37
arch-x86-kernel-cpu-microcode-microcode.c	129_1a	47.41	8.93	17	0	TRUE	TRUE	0.0	5.31
drivers-vhost-vhost_net.c	32_1a	129.45	25.48	1080	68	TRUE	TRUE	0.06	5.08
drivers-staging-media-lirc-lirc_zilog.c	32_1a	124.62	27.29	336	14	TRUE	TRUE	0.04	4.57
net-phonet-phonet.c	32_1a	877.49	193.37	9716	1088	UNKN	TRUE	0.11	4.54
drivers-staging-media-bcm2048-radio-bcm2048.c	32_1a	105.76	24.62	1386	13	TRUE	TRUE	0.01	4.3
drivers-net-ethernet-microchip-enc28j60.c	32_1a	191.99	45.25	164	3	TRUE	TRUE	0.02	4.24
drivers-usb-atm-cxacru.c	32_1a	120.66	28.51	283	22	TRUE	TRUE	0.08	4.23
drivers-thunderbolt-thunderbolt.c	32_1a	322.9	79.07	1081	70	TRUE	TRUE	0.06	4.08
drivers-net-wireless-ath-carl9170-carl9170.c	43_1a	802.46	197.29	1368	0	TRUE	TRUE	0.0	4.07
drivers-video-fbdev-matrox-matroxfb_base.c	32_1a	879.78	224.6	2681	29	UNKN	TRUE	0.01	3.92
drivers-platform-x86-samsung-laptop.c	32_1a	813.33	207.72	173	3	TRUE	TRUE	0.02	3.92
drivers-mfd-sm501.c	32_1a	880.72	229.05	3744	406	UNKN	TRUE	0.11	3.85
drivers-input-serio-serio.c	43_1a	880.87	236.36	12 267	0	UNKN	TRUE	0.0	3.73
drivers-platform-x86-acer-wmi.c	32_1a	89.8	24.32	7	1	TRUE	TRUE	0.14	3.69
drivers-scsi-scsi_debug.c	129_1a	878.71	242.8	378	0	UNKN	TRUE	0.0	3.62
drivers-hwmon-w83627ehf.c	32_1a	372.2	105.58	373	22	TRUE	TRUE	0.06	3.53
drivers-vhost-vhost_scsi.c	32_1a	79.26	22.89	1280	101	TRUE	TRUE	0.08	3.46
drivers-net-wireless-rtlwifi-rtl_pci.c	43_1a	877.21	254.91	7	0	UNKN	TRUE	0.0	3.44
drivers-mtd-mtd_blkdevs.c	32_1a	164.96	51.16	2266	203	TRUE	TRUE	0.09	3.22
drivers-media-usb-dvb-usb-v2-dvb-usb-af9015.c	32_1a	273.04	84.76	620	16	TRUE	TRUE	0.03	3.22
drivers-watchdog-pcwd_usb.c	32_1a	95.4	29.78	175	26	TRUE	TRUE	0.15	3.2

Program	Spec.	An. CPU time		# Refinements		Result		Ref. used	Speedup
		no	yes	no	yes	no	yes		
drivers-net-macvlan.c	129_1a	25.84	8.13	36	0	TRUE	TRUE	0.0	3.18
net-bridge-netfilter-ebtables.c	129_1a	56.04	17.95	141	0	TRUE	TRUE	0.0	3.12
drivers-media-usb-dvb-usb-v2-dvb-usb-lmedm04.c	32_1a	57.68	18.64	14	2	FALSE	FALSE	0.14	3.09
drivers-hwmon-pc87360.c	32_1a	179.02	58.92	2775	35	TRUE	TRUE	0.01	3.04
drivers-net-wireless-rt2x00-rt61pci.c	32_1a	878.62	301.43	726	21	UNKN	TRUE	0.03	2.91
drivers-staging-lustre-lustre-mgc-mgc.c	32_1a	108.51	37.33	141	3	TRUE	TRUE	0.02	2.91
drivers-infiniband-core-rdma_ucm.c	32_1a	58.24	20.78	469	17	TRUE	TRUE	0.04	2.8
drivers-net-wan-hdlc_ppp.c	43_1a	51.67	18.76	36	0	TRUE	TRUE	0.0	2.75
drivers-net-ethernet-fealnx.c	43_1a	27.23	10.09	56	0	TRUE	TRUE	0.0	2.7
drivers-hid-hid-roccat.c	32_1a	49.33	18.4	346	9	TRUE	TRUE	0.03	2.68
drivers-net-ethernet-renesas-sh_eth.c	43_1a	173.38	64.71	53	0	TRUE	TRUE	0.0	2.68
drivers-mmc-host-vub300.c	32_1a	163.15	61.23	387	126	TRUE	TRUE	0.33	2.66
drivers-nfc-trf7970a.c	32_1a	55.73	21.49	172	1	TRUE	TRUE	0.01	2.59
net-packet-af_packet.c	32_1a	345.41	133.26	1026	295	TRUE	TRUE	0.29	2.59
drivers-media-radio-radio-si476x.c	32_1a	135.45	53.2	987	2	TRUE	TRUE	0.0	2.55
net-rxrpc-rxkad.c	32_1a	187.3	73.97	92	18	TRUE	TRUE	0.2	2.53
drivers-net-ethernet-altera-altera_tse.c	43_1a	119.99	48.24	19	1	TRUE	TRUE	0.05	2.49
drivers-hid-hid-logitech-hidpp.c	32_1a	719.32	293.62	455	160	TRUE	TRUE	0.35	2.45

Table A.2: Median results

Program	Spec.	An. CPU time		# Refinements		Result		Ref. used	Speedup
		no	yes	no	yes	no	yes		
drivers-media-radio-si4713-radio-usb-si4713.c	32_1a	15.95	15.36	16	6	TRUE	TRUE	0.38	1.04
drivers-hwmon-adcxx.c	32_1a	8.93	8.6	13	5	TRUE	TRUE	0.38	1.04
drivers-iio-dac-ad5360.c	32_1a	13.95	13.44	43	18	TRUE	TRUE	0.42	1.04
drivers-cpufreq-cpufreq_stats.c	43_1a	12.6	12.14	5	5	TRUE	TRUE	1.0	1.04
drivers-iio-adc-nau7802.c	32_1a	10.41	10.03	6	2	TRUE	TRUE	0.33	1.04
sound-usb-usx2y-snd-usb-us122l.c	32_1a	27.99	26.97	187	91	TRUE	TRUE	0.49	1.04
drivers-iio-light-apds9300.c	32_1a	12.44	11.99	26	1	TRUE	TRUE	0.04	1.04
sound-usb-snd-usbmidi-lib.c	32_1a	25.3	24.39	138	82	TRUE	TRUE	0.59	1.04
net-ipv4-udp_diag.c	43_1a	11.96	11.53	3	3	TRUE	TRUE	1.0	1.04
drivers-memstick-host-rtsx_pci_ms.c	32_1a	9.61	9.27	10	3	TRUE	TRUE	0.3	1.04
drivers-iio-light-isl29125.c	32_1a	11.6	11.19	15	6	TRUE	TRUE	0.4	1.04
sound-pci-snd-als300.c	43_1a	19.07	18.4	4	4	TRUE	TRUE	1.0	1.04
drivers-i2c-i2c-dev.c	43_1a	17.25	16.65	9	9	TRUE	TRUE	1.0	1.04
drivers-iio-magnetometer-mag3110.c	32_1a	10.37	10.01	14	3	TRUE	TRUE	0.21	1.04
mm-zsmalloc.c	43_1a	24.8	23.94	21	17	TRUE	TRUE	0.81	1.04
drivers-w1-slaves-w1_ds2408.c	32_1a	8.09	7.81	7	1	TRUE	TRUE	0.14	1.04
drivers-block-loop.c	43_1a	28.13	27.16	5	5	TRUE	TRUE	1.0	1.04
drivers-usb-misc-ftdi-elan.c	134_1a	22.45	21.68	5	5	TRUE	TRUE	1.0	1.04
drivers-net-wireless-zd1201.c	68_1a	17.57	16.97	12	8	FALSE	FALSE	0.67	1.04
drivers-staging-media-dt3155v4l-dt3155v4l.c	32_1a	11.14	10.76	3	2	FALSE	FALSE	0.67	1.04
net-ipv4-esp4.c	43_1a	14.98	14.47	2	1	TRUE	TRUE	0.5	1.04
drivers-staging-iio-resolver-ad2s1210.c	32_1a	12.7	12.27	10	3	TRUE	TRUE	0.3	1.04
drivers-isdn-hisax-hisax_st5481.c	43_1a	229.96	222.23	198	41	TRUE	TRUE	0.21	1.03
drivers-net-slip-slip.c	43_1a	18.02	17.42	20	5	TRUE	TRUE	0.25	1.03
drivers-net-ethernet-atheros-atlx-atl1.c	43_1a	226.14	218.62	295	59	TRUE	TRUE	0.2	1.03
drivers-usb-host-fotg210-hcd.c	129_1a	82.72	79.98	39	0	TRUE	TRUE	0.0	1.03
drivers-edac-i7core_edac.c	32_1a	16.33	15.79	48	22	TRUE	TRUE	0.46	1.03
drivers-usb-serial-ir-usb.c	43_1a	12.27	11.87	5	3	TRUE	TRUE	0.6	1.03
drivers-block-nbd.c	43_1a	8.7	8.42	1	0	TRUE	TRUE	0.0	1.03
drivers-tty-serial-max310x.c	32_1a	25.43	24.62	106	61	TRUE	TRUE	0.58	1.03
drivers-mtd-chips-cfi_cmdset_0002.c	32_1a	9.43	9.13	6	3	TRUE	TRUE	0.5	1.03
drivers-watchdog-wm831x_wdt.c	32_1a	10.7	10.36	29	3	TRUE	TRUE	0.1	1.03
drivers-tty-serial-mfd.c	43_1a	95.12	92.11	52	51	TRUE	TRUE	0.98	1.03
drivers-usb-misc-usbtest.c	32_1a	38.77	37.55	36	21	FALSE	FALSE	0.58	1.03
drivers-media-common-siano-smsdvb.c	32_1a	13.35	12.93	7	6	TRUE	TRUE	0.86	1.03
drivers-net-wan-dscc4.c	32_1a	126.36	122.4	211	205	TRUE	TRUE	0.97	1.03
drivers-scsi-cxgbi-cxgb3i-cxgb3i.c	43_1a	21.79	21.11	1	0	TRUE	TRUE	0.0	1.03
net-sctp-sctp_probe.c	43_1a	15.39	14.91	4	4	TRUE	TRUE	1.0	1.03
drivers-media-pci-solo6x10-solo6x10.c	32_1a	27.98	27.11	10	9	FALSE	FALSE	0.9	1.03
net-ipv4-netfilter-arp_tables.c	08_1a	15.78	15.29	8	5	FALSE	FALSE	0.62	1.03
net-core-drop_monitor.c	32_1a	11.3	10.95	12	6	TRUE	TRUE	0.5	1.03
drivers-media-usb-dvb-usb-v2-dvb-usb-az6007.c	32_1a	10.35	10.03	3	3	FALSE	FALSE	1.0	1.03
drivers-iio-dac-ad5686.c	32_1a	11.66	11.3	13	3	TRUE	TRUE	0.23	1.03
drivers-media-pci-pt1-earth-pt1.c	32_1a	21.43	20.78	68	19	TRUE	TRUE	0.28	1.03
sound-drivers-snd-dummy.c	43_1a	743.42	721.06	8	8	TRUE	TRUE	1.0	1.03
sound-pci-snd-azt3328.c	43_1a	22.0	21.34	2	2	TRUE	TRUE	1.0	1.03
drivers-net-sb1000.c	134_1a	109.35	106.07	14	14	TRUE	TRUE	1.0	1.03
drivers-media-radio-si470x-radio-usb-si470x.c	68_1a	74.44	72.23	27	25	FALSE	FALSE	0.93	1.03
drivers-usb-storage-ums-realtek.c	32_1a	16.06	15.59	36	17	TRUE	TRUE	0.47	1.03
drivers-net-hamradio-mkiss.c	43_1a	21.92	21.29	38	14	TRUE	TRUE	0.37	1.03

Table A.3: Bottom results

Program	Spec.	An. CPU time		# Refinements		Result		Ref. used	Speedup
		no	yes	no	yes	no	yes		
drivers-md-dm-log.c	43_1a	8.33	9.54	1	1	TRUE	TRUE	1.0	0.87
drivers-media-pci-ddbridge-ddbridge.c	43_1a	9.47	10.85	1	1	TRUE	TRUE	1.0	0.87
drivers-edac-amd64_edac_mod.c	129_1a	8.99	10.37	1	0	TRUE	TRUE	0.0	0.87
drivers-iio-imu-inv_mpu6050-inv-mpu6050.c	08_1a	10.89	12.61	1	1	FALSE	FALSE	1.0	0.86
drivers-usb-atm-xusbatm.c	134_1a	6.81	7.89	2	2	TRUE	TRUE	1.0	0.86
drivers-iio-gyro-itg3200.c	08_1a	9.95	11.61	1	1	FALSE	FALSE	1.0	0.86
drivers-media-rc-img-ir-img-ir.c	32_1a	178.29	208.73	8	2	TRUE	TRUE	0.25	0.85
drivers-usb-serial-mos7840.c	43_1a	37.01	43.33	200	28	TRUE	TRUE	0.14	0.85
drivers-input-joystick-xpad.c	68_1a	54.95	64.83	28	30	FALSE	FALSE	1.07	0.85
drivers-net-ethernet-marvell-pxa168_eth.c	43_1a	20.96	25.11	26	14	TRUE	TRUE	0.54	0.83
drivers-mfd-tps65010.c	32_1a	16.02	19.55	69	9	TRUE	TRUE	0.13	0.82
sound-soc-intel-snd-soc-sst-baytrail-pcm.c	43_1a	10.31	12.63	2	0	TRUE	TRUE	0.0	0.82
drivers-net-wireless-rtlwifi-rtl_usb.c	43_1a	51.13	62.7	527	13	TRUE	TRUE	0.02	0.82
drivers-usb-class-cdc-wdm.c	43_1a	25.26	30.98	45	32	TRUE	TRUE	0.71	0.82
drivers-net-wireless-ti-wl12xx-wl12xx.c	32_1a	179.2	219.94	3	3	FALSE	FALSE	1.0	0.81
drivers-net-ethernet-sun-sunhme.c	43_1a	327.74	407.52	308	51	TRUE	TRUE	0.17	0.8
sound-drivers-snd-portman2x4.c	43_1a	10.74	13.39	1	1	TRUE	TRUE	1.0	0.8
drivers-net-wan-dscc4.c	43_1a	38.75	50.06	92	48	TRUE	TRUE	0.52	0.77
drivers-atm-zatm.c	43_1a	72.1	93.27	39	41	TRUE	TRUE	1.05	0.77
net-key-af_key.c	32_1a	658.28	854.05	193	41	TRUE	TRUE	0.21	0.77
drivers-usb-gadget-function-u_ether.c	43_1a	319.77	417.13	4	0	TRUE	TRUE	0.0	0.77
drivers-net-ethernet-ti-tlan.c	134_1a	67.98	89.75	2	12	TRUE	TRUE	6.0	0.76
drivers-input-tablet-gtco.c	68_1a	665.05	880.66	62	58	FALSE	UNKN	0.94	0.76
drivers-atm-iphase.c	43_1a	182.22	243.66	58	58	TRUE	TRUE	1.0	0.75
drivers-isdn-hisax-hisax_st5481.c	68_1a	550.66	739.73	48	41	FALSE	FALSE	0.85	0.74
drivers-usb-class-cdc-acm.c	43_1a	67.33	90.62	393	392	TRUE	TRUE	1.0	0.74
drivers-md-dm-snapshot.c	43_1a	7.99	10.98	1	1	TRUE	TRUE	1.0	0.73
drivers-net-ethernet-micrel-ks8842.c	43_1a	30.37	42.08	79	15	TRUE	TRUE	0.19	0.72
sound-core-seq-oss-snd-seq-oss.c	43_1a	383.55	552.02	96	93	TRUE	TRUE	0.97	0.69
drivers-net-usb-kaweth.c	134_1a	15.97	23.44	1	7	TRUE	TRUE	7.0	0.68
drivers-usb-serial-garmin_gps.c	43_1a	39.14	57.8	129	8	TRUE	TRUE	0.06	0.68
drivers-bluetooth-btusb.c	68_1a	45.33	67.26	9	10	TRUE	TRUE	1.11	0.67
drivers-infiniband-core-ib_mad.c	43_1a	77.2	116.34	34	25	TRUE	TRUE	0.74	0.66
drivers-usb-serial-digi_acceleport.c	43_1a	25.13	38.71	112	11	TRUE	TRUE	0.1	0.65
drivers-media-dvb-frontends-rtl2832_sdr.c	43_1a	35.95	63.16	70	15	TRUE	TRUE	0.21	0.57
drivers-net-hippi-rrunner.c	43_1a	26.48	47.1	23	25	TRUE	TRUE	1.09	0.56
drivers-net-ethernet-tehuti-tehuti.c	43_1a	251.23	500.68	352	212	TRUE	TRUE	0.6	0.5
drivers-net-vmxnet3-vmxnet3.c	43_1a	202.1	425.12	582	126	TRUE	TRUE	0.22	0.48
drivers-net-ethernet-cadence-macb.c	43_1a	80.37	171.67	101	100	TRUE	TRUE	0.99	0.47
drivers-net-ethernet-micrel-ksz884x.c	43_1a	406.71	876.92	1382	886	TRUE	UNKN	0.64	0.46
drivers-dma-mic_x100_dma.c	43_1a	22.3	48.15	19	22	TRUE	TRUE	1.16	0.46
drivers-usb-mon-usbmon.c	43_1a	15.14	34.47	4	5	TRUE	TRUE	1.25	0.44
drivers-net-irda-vlsi_ir.c	43_1a	26.07	62.76	31	24	TRUE	TRUE	0.77	0.42
drivers-net-ethernet-atheros-atl1c-atl1c.c	43_1a	335.36	877.11	143	63	TRUE	UNKN	0.44	0.38
drivers-net-ethernet-sis-sis900.c	43_1a	206.72	584.94	257	469	TRUE	TRUE	1.82	0.35
drivers-tty-n_r3964.c	43_1a	284.66	880.57	105	448	TRUE	UNKN	4.27	0.32
drivers-net-wireless-orinoco-orinoco_usb.c	43_1a	17.62	62.38	40	4	TRUE	TRUE	0.1	0.28
drivers-usb-serial-cypress_m8.c	43_1a	239.38	883.35	73	2	TRUE	UNKN	0.03	0.27
drivers-atm-nicstar.c	43_1a	212.92	878.03	1376	207	TRUE	UNKN	0.15	0.24
drivers-isdn-capi-capidrv.c	43_1a	14.51	120.36	21	0	TRUE	TRUE	0.0	0.12

Bibliography

[1] Sven Apel et al. "On-the-fly Decomposition of Specifications in Software Model Checking". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. New York, NY, USA: ACM, 2016, pp. 349–361.

[2] Thomas Ball et al. "Automatic Predicate Abstraction of C Programs". In: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. ACM, 2001, pp. 203–213.

[3] D. Beyer, T. A. Henzinger, and G. Theoduloz. "Program Analysis with Dynamic Precision Adjustment". In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 2008, pp. 29–38.

[4] D. Beyer, M. E. Keremoglu, and P. Wendler. "Predicate abstraction with adjustable-block encoding". In: *Formal Methods in Computer Aided Design*. Oct. 2010, pp. 189–197.

[5] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis". In: *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings*. Springer, 2007, pp. 504–518.

[6] Dirk Beyer and M. Erkan Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011, Snowbird, UT, July 14-20)*. Ed. by G. Gopalakrishnan and S. Qadeer. LNCS 6806. Springer-Verlag, Heidelberg, 2011, pp. 184–190.

[7] Dirk Beyer and Stefan Löwe. "Explicit-State Software Model Checking Based on CEGAR and Interpolation". In: *Fundamental Approaches to Software Engineering: 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Springer, 2013, pp. 146–162.

[8] Dirk Beyer et al. "Precision Reuse for Efficient Regression Verification". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 389–399.

[9] Edmund Clarke et al. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings*. Springer, 2000, pp. 154–169.

[10] Susanne Graf and Hassen Saidi. "Construction of abstract state graphs with PVS". In: *Computer Aided Verification: 9th International Conference, CAV'97 Haifa, Israel, June 22–25, 1997 Proceedings*. Springer, 1997, pp. 72–83.

[11] Thomas A. Henzinger et al. "Abstractions from Proofs". In: *SIGPLAN Not.* 39.1 (2004), pp. 232–244.

[12] Ranjit Jhala and Rupak Majumdar. "Software Model Checking". In: *ACM Comput. Surv.* 41.4 (2009), 21:1–21:54.

[13] K. L. McMillan. "Interpolation and SAT-Based Model Checking". In: *Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings*. Springer, 2003, pp. 1–13.

[14] Saurabh Srivastava and Sumit Gulwani. "Program Verification Using Templates over Predicate Abstraction". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. ACM, 2009, pp. 223–234.

[15] Margus Veanes. "Applications of Symbolic Finite Automata". In: *Implementation and Application of Automata: 18th International Conference, CIAA 2013, Halifax, NS, Canada, July 16-19, 2013. Proceedings*. Springer, 2013, pp. 16–23.

Eidesstattliche Erklärung:

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

Passau, den 29. September 2017 ______________________________

Sebastian Böhm