UNIVERSITÄT PASSAU
*Fakultät für Informatik und Mathematik*

University of Passau
Faculty of Computer Science and Mathematics
Chair of Software Engineering I

# A Provident Abstraction Refinement Framework for Multi-Path Refinements

Master Thesis in Computer Science

2018–03–08

**Stephan Lukasczyk**

| | |
|---|---|
| Examiners: | Prof. Dr.-Ing. Sven Apel<br>(Chair of Software Engineering I) |
| | Prof. Dr. Gordon Fraser<br>(Chair of Software Engineering II) |
| Tutor: | Andreas Stahlbauer, M. Sc.<br>(Chair of Software Engineering I) |

**Lukasczyk, Stephan**:
*A Provident Abstraction Refinement Framework for Multi-Path Refinements*
Master Thesis, University of Passau, 2018.



**Abstract**

We propose a generalised and configurable abstraction refinement framework for provident and prescient refinements. Our procedure therefore calculates the information gained from multiple paths to multiple error locations in a provident way. This means, it only calculates the changes caused by the refinement but does not perform any changes on the underlying abstraction, yet. The gained information, that is, the precision update necessary to rule out a specific (possibly) infeasible error path, can be used to come up with the best suiting refinement for the current verification problem.

The provident refinement procedure is able to delay, estimate, combine, and rank the possible refinements based on the calculated information about their precisions. To achieve these goals, we developed an extended, CEGAR-based, algorithm that allows us to implement our provident refinement procedure.

Furthermore, we provide an empirical study on the performance characteristics of our proposed technique using 250 kernel driver modules from the Linux 4.0-rc1 kernel and 14 safety properties modelling the correct behaviour and correct API usage of those driver modules.

# Contents

# List of Terms and Abbreviations

**API** Application Programming Interface

**ARG** Abstract Reachability Graph

BenchExec A framework for reliable benchmarking and resource measurements, based on cgroups and built in Python

**CEGAR** Counterexample-Guided Abstraction Refinement

**CFA** Control-Flow Automaton

**CPA** Configurable Program Analysis

CPAchecker An open source tool and framework for software verification and program analysis

**JIT** Just-in-Time

**JVM** Java Virtual Machine

**LDV** Linux Driver Verification

MathSAT A SMT solver supporting a wide range of theories and functionalities

**PCEGAR** Provident Counterexample-Guided Abstraction Refinement

# List of Algorithms



# List of Figures



# List of Tables

# 1 Introduction

> If your specifications are ambiguous, the greater the ambiguity, the easier the specifications are to satisfy (if the specifications are absolutely ambiguous, every program will satisfy them!).
>
> *(Edsger W. Dijkstra [Dij74])*

Every non-trivial piece of software contains bugs. Steve McConnell states that the industry average lies between 15 and 50 errors per 1 000 lines of delivered code [McC04]. By using special development techniques it is possible to achieve rates between 0.1 and 3 defects per 1 000 lines of code [CM90]. Compared to the millions of lines of code, for example, current operating systems consist of, it follows that there are thousands to ten-thousand defects in such a piece of software.

## 1.1. Motivation

Therefore, techniques have been developed to reduce the number of such errors or even mathematically prove that errors are absent. One such well-established technique for formal verification is model checking [CES83; CES86]. By using an abstract model of the system one avoids the complexity of the original system during the process. The model is then checked against a formal specification that, for example, models the correct behaviour of a software system. In case the specification does not hold, a counterexample that describes the violation, is produced.

Real-world software is complex, thus it is almost infeasible to construct an exact model of the program. Hence, the usage of abstraction techniques is a requirement when building the model. This makes the abstract model small but has a major drawback: it is possible that the abstract model is too coarse, that is, it does not contain enough information to prove the correctness of the program regarding the specification.

Because of this it is possible that the model checker yields a *false* verdict, that means, it detects a violation of the specification in the abstract model, although this violation is not possible in the original system. In order to keep the abstract model small without getting any false alarms,

the abstraction needs to be refined to rule out the infeasible counterexample. A well-known and well-established technique to achieve this is called *Counterexample-Guided Abstraction Refinement* (CEGAR) [CES83; CES86].

In the CEGAR loop a found counterexample will be checked whether it is real or spurious. A spurious counterexample means that the abstract model is too coarse [CGJ+03]. Thus, a refinement of the abstract model is needed to remove the found counterexample. For that, the information from the found path will be used, for example, Craig interpolants for predicate-based analyses.

Often there is more than one refinement possible as there is more than one path through the program to an error location or even more than one error location. The selection of a refinement to apply to the abstract model, in order to remove a spurious counterexample, is crucial. The quality of the information that is gained from the counterexample influences the quality of the analysis [BLW15c]. Many of today's approaches are based on lazy abstraction [HJM+02] and thus perform a refinement each time an error location is found. This obviously limits the possible range to select a refinement, as the order of found error locations is only determined by the state-space exploration strategy, for example, depth-first or breadth-first search on a control-flow graph. While refinement strategies based on lazy abstraction work well in practice, this can be limiting in case many properties are check checked at once, as it is the case, for example, with multi-property verification [ABM+16].

Previous work already identified that there is often more than one reason for the infeasibility of a spurious counterexample and proposed heuristics to select the best-suiting refinement [BLW15b]. Nevertheless, they only consider single paths to single error locations. Targeting this problem is the main goal of our work. We want to incorporate the information of all possible refinements. Although, we still rely on the state-space exploration strategy, this allows us to consider more information for our selection, in order to decrease the number of necessary refinements and gain performance.

Furthermore, previous work often bases their evaluation on artificial benchmark sets or benchmark sets that are specifically created to exhibit the proposed behaviour of a new approach. Hence, to avoid such artificial benchmarks, we use a case study from a real-world software project, namely the Linux kernel. We combine it with a set of safety properties that model the correct usage of some of the kernel's *Application Programming Interfaces* (APIs).

## 1.2. Contributions

With our work, we provide the following contributions:

- We define a provident multi-path refinement framework, that is, a framework that takes paths to more than one target state into

account. It is able to use information about the target states and the calculated refinement candidates to rank these refinements and combine them in, order to apply the best-suiting refinement for the given verification task.

- We implement our approach in a custom fork of the software verification framework CPAchecker [1]. The framework is an open source project and available under the terms of the Apache 2.0 license [BHT07; BHT08]. The code of the fork is available in a (currently private) GitHub repository.
- We evaluate our multi-path refinement strategy and compare it to a traditional single-path strategy in order to gain information about the performance behaviour of a multi-path strategy. The evaluation is done using a case study built of 250 Linux kernel modules together with a set of 14 safety properties. The case study was also used in prior work [ABM+16] and was created in collaboration with the *Linux Driver Verification* (LDV) project [KMP+09].

All data, as well as the case study, can be found on our supplementary web page[2].

## 1.3. Structure of the Thesis

Before we introduce our novel refinement framework (Chapter 3), we give an overview over the relevant background (Chapter 2) that is necessary to understand the concept we use later on. In order to show the effects of our concept, we provide an extensive evaluation (Chapter 4). We shall provide an overview over related work (Chapter 5) and conclude our presentation with an outlook on possible future research directions (Chapter 6).

[1] See https://cpachecker.sosy-lab.org for more information.
[2] See https://research.lukasczyk.me/provident-refinement

# 2

# Background

We present the relevant background for our work in this chapter. We explain the ideas of software model checking to the reader, give an insight in the program representation we use, and introduce the CPAchecker framework. Furthermore, we introduce CEGAR, precision adjustment, and Craig interpolation. You might want to skip this chapter if you are familiar with the presented concepts.

## 2.1. Software Model Checking

*Software model checking* is the algorithmic analysis of programs to prove properties of the executions [JM09]. All information of this section is taken from [JM09].

Model checking has its roots in logic and theorem proving. It is a conceptual framework for the formalisation of verification problems and provides algorithmic procedures for the analysis. Providing a sound and complete algorithmic solution to the problem is generally infeasible due to Turing's undecidability theorem [Tur37]. Research on verification starts with manual reasoning (see for example [Hoa69]). Due to the increasing complexity of the verification problems automatic tools became necessary. In the beginning, those tools needed human guidance, such as the provision of loop invariants or pre- and post-conditions [Dij76]. Modern tools often combine several different techniques such as theorem proving, model checking, or dataflow analyses.

Model checking tends to prove properties of program computations. Such properties can, for example, be assertions, like a variable that is guaranteed to have a certain value at a certain program location; the property can also be a global invariant or a termination property. A *safety property* can be expressed by the reachability of a certain program location, the *error location*. By combining an input program $P$ and a property $\Pi$ a *verification problem* is built. The model checker returns "safe" if every computation of $P$ is in $\Pi$; otherwise it returns "unsafe".

The path from the initial program location to the error location is called counterexample or error path.

Jhala and Majumdar provide an extensive survey [JM09] on techniques and tools for software model checking.

In this section we provide basic definitions taken from the literature [BDW18] on the representation of programs we use. We restrict our presentation to a simple imperative programming languages that only consists of assignments and assumptions, where all variables range over integers[1].

We represent a program as a *Control-Flow Automaton* (CFA), which is a directed graph with program operations attached to its edges. A set $L$ of *program locations*, an initial location $l_{\text{INIT}}$, which represents the program entry point, a set $Ops$ of *program operations*, and a set $G \subseteq (L \times Ops \times L)$ of edges between program locations forms a CFA $A = (L, l_{\text{INIT}}, G)$. Each edge is labelled with a program operation that is executed, when the control flow walks along the edge. We denote the set of all program variables by $X$; these are the variables that occur in operations of a CFA. A *concrete data state* $c : X \to \mathbb{Z}$ is a mapping from program variables to integers. A set of such concrete states is called *region* and represented by first-order formulæ $\psi$ over variables from $X$ such that the set $[\![\psi]\!]$ of concrete data states that is represented by $\psi$ is defined as $[\![\psi]\!] = \{c : c \models \psi\}$. A pair $(c, l) : (X \to \mathbb{Z}) \times L$ of a concrete data state and a location is called *concrete state*.

Each operation $op \in Ops$ can either be an assignment of the form $x := e$, consisting of a variable $x \in X$ and a side-effect free arithmetic expression $e$ over variables from $X$, or an assume operation $[p]$, consisting of a predicate $p$ over variables from $X$. The *strongest-postcondition operator* $\mathsf{SP}_{op}(\cdot)$ defines the semantics of an operation $op$. For a formula $\psi$ and an assignment $x := e$ the strongest-postcondition operator is defined as $\mathsf{SP}_{x:=e}(\psi) = \exists \hat{x} : \psi_{[x \to \hat{x}]} \wedge (x = e_{[x \to \hat{x}]})$; for an assume operation $[p]$ it is defined as $\mathsf{SP}_{[p]}(\psi) = \psi \wedge p$. The existential quantifier in the strongest-postcondition operator for assignments can be avoided in the implementation by skolemization.

We call a sequence of consecutive edges from $G$ a *path*, denoted by $\sigma = \langle (l_i, op_i, l_j), (l_j, op_j, l_k), \dots, (l_m, op_m, l_n) \rangle$. If a path starts in the initial location $l_{\text{INIT}}$, we call it *program path*. By the iterative application of $\mathsf{SP}_{op}(\cdot)$ we define the semantics of a path for each operation of the path: $\mathsf{SP}_{\sigma}(\psi) = \mathsf{SP}_{op_m}(\dots(\mathsf{SP}_{op_i}(\psi))\dots)$. If $\mathsf{SP}_{\sigma}(\mathit{true})$ is satisfiable we call a path $\sigma$ *feasible*; otherwise it is called *infeasible*. We call a location $l$ *reachable* if there exists a feasible path from $l_{\text{INIT}}$ to $l$.

Listing 2.1 shows a small C program and Figure 2.1 a corresponding CFA. The program contains a `while`-loop running a non-deterministic number of iterations—given by the `nondet()` function. It contains a reachable `ERROR` label, which indicates a specification violation.

[1] Our implementation, however, supports C programs.

```c
extern int
  ↪  nondet(void);

int main(void) {
  int x = nondet();
  int i = nondet();

  while (i) {
    ++x;

    if (x % 2 == 0)
      goto ERROR;
    else
      i = nondet();

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

**Listing 2.1.** A simple example program in C

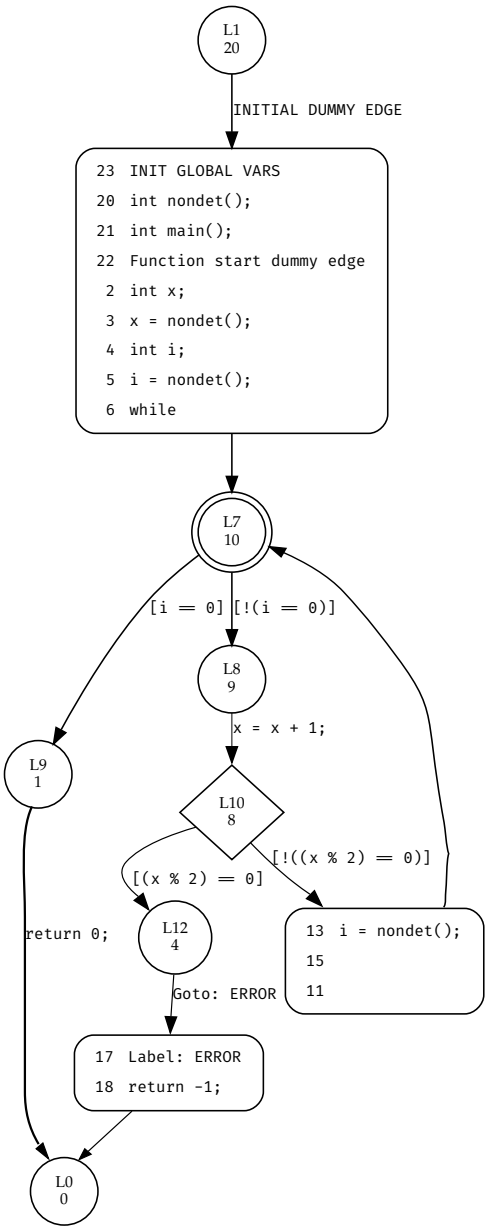


**Figure 2.1.** A CFA for the program in Listing 2.1 as generated by CPAchecker

## 2.3. CPAchecker

CPAchecker is a tool and framework for software verification, especially for C programs [BK11]. It is an open-source tool written in Java and available under the terms of the Apache 2.0 license. More on the framework and its documentation can be found on the project's web site[2].

[2] See https://cpachecker.sosy-lab.org

### 2.3.1. Configurable Program Analysis

The core concept of CPAchecker is based on *Configurable Program Analysis* (CPA) [BHT07] with dynamic precision adjustment [BHT08]. We take the following definitions from the literature [ABM+16; BDW18].

A CPA with dynamic precision adjustment (note that this precisely is called CPA+, but it is common to refer to CPA+ as CPA if the difference is not important for the presentation) is of the form

$$\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec}).$$

The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ defines the type of abstract representation of concrete states from $C$. It consists of the set of all concrete states $C$ of the program, a semi-lattice $\mathcal{E} = (E, \sqsubseteq, \sqcup, \top)$, and a concretisation function $[\![\cdot]\!]$ that maps each abstract-domain element to the represented set of concrete states. $E$ denotes the set of abstract-domain elements, this means, the abstract states; $\sqsubseteq$ denotes the lattice's partial order. An abstract state $e \in E$ is called *abstract error state* if it represents a concrete state at the error location $l_{\text{ERR}}$, that is, if $\exists c \in (X \to \mathbb{Z}) : (c, l_{\text{ERR}}) \in [\![e]\!]$; we also denote it by $e_{\text{ERR}}$.

The *precision* $\pi \in \Pi$ defines aspects of the state space that should be represented by abstract states in a given abstract domain. The set of successor abstract states for each abstract state is defined by the *transfer relation* $\rightsquigarrow \subseteq E \times E \times \Pi$. The merge operator $\mathsf{merge} : E \times E \times \Pi \to E$ provides the ability to merge two abstract states under a given precision. To determine whether an abstract state is covered by other abstract states, the stop operator $\mathsf{stop} : E \times 2^E \times \Pi \to \mathbb{B}$ is used. Finally, the precision-adjustment operator $\mathsf{prec} : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$ allows a dynamic adjustment of the precision depending on the current set of reachable abstract states.

Such CPAs are the central building blocks of this formalism. Several CPAs are composed to form a *composite analysis*. We refer the reader to the literature [BHT07; BHT08; BK11; BDW18] for further details on the CPA formalism.

### 2.3.2. The CPA Algorithm

The CPA algorithm (see Algorithm 2.1) uses a combination of CPAs and an initial abstract state with precision for a reachability analysis and performs a classic fixed-point iteration by looping until the set waitlist is empty, that is, all abstract states have been completely processed. The algorithm returns the set of reachable abstract states (denoted by reached). In each algorithm iteration, it takes one abstract state $e$ with precision $\pi$ from the waitlist, executes the precision-adjustment operator prec, computes all abstract successors, and processes each of those successors [BDW18].

We refer the reader to the literature [BHT07; BHT08; BK11; BDW18] for further details.

**Algorithm 2.1:** The CPA++($\mathbb{D}$, reached, waitlist, abort) algorithm, taken from [BDW18]

**Input:** a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \text{merge}, \text{stop}, \text{prec})$, with additional operator fcover, where $E$ denotes the set of elements of the semilattice of $D$,
a set reached $\in E \times \Pi$ of reachable abstract states
a set waitlist $\in E \times \Pi$ of frontier abstract states, and
a function abort : $E \to \mathbb{B}$ that defines whether the algorithm should abort early

**Output:** the updated sets reached and waitlist

```
while waitlist ≠ ∅ do
    pop (e, π) from waitlist;
    reached:= fcover(reached, e, π);
    if (e, π) ∉ reached then
        continue
    forall e' with e ⇝ (e', π) do
        (ê, π̂) := prec(e', π, reached);
        forall (e'', π'') ∈ reached do
            e_new := merge(ê, e'', π̂);
            if e_new ≠ e'' then
                waitlist := (waitlist ∪ {(e_new, π̂)}) \ {(e'', π'')};
                reached := (reached ∪ {(e_new, π̂)}) \ {(e'', π'')};
        if not stop(ê, {e : (e, ·) ∈ reached}, π̂) then
            waitlist := waitlist ∪ {(ê, π̂)};
            reached := reached ∪ {(ê, π̂)};
            if abort (ê) then
                return (reached, waitlist)
return (reached, waitlist)
```

### 2.3.3. *Basic CPAs*

We introduce some CPAs, which we refer to in the later presentation. The following is taken from [BDW18].

As mentioned before, it is possible to combine several CPAs using a *Composite CPA* [BHT07] in the style of the well-known Composite Pattern [GHJ+95]. This allows to seperate concerns into different CPAs; they can then be reused and flexibly combined to create new analyses, instead of redefining them for every analysis.

Most kinds of program analysis need to track the program counter, for example. The *Location CPA* $\mathbb{L}$ [BHT08] tracks the program counter and allows other CPAs to use the information without implementing a program counter tracking on their own.

Furthermore, we define an additional *ARG CPA* $\mathbb{A}$ that tracks the *Abstract Reachability Graph* (ARG) over the abstract states in the set reached. It stores the predecessor-successor relationship between abstract states and allows to reconstruct abstract paths in the ARG. We call a se-

quence $\langle e_0, \ldots, e_n \rangle$ of abstract states an *abstract path* if for any pair $(e_i, e_{i+1})$ with $i \in \{0, \ldots, n-1\}$ either $e_{i+1}$ is an abstract successor of $e_i$ or $e_{i+1}$ is the result of merging an abstract successor of $e_i$ with some other abstract state(s). The combination of both Location CPA and ARG CPA allows us to reconstruct from an abstract path the path that it represents in the CFA. Unrolling the paths through the ARG forms an abstract reachability tree.

## 2.4. Counterexample-Guided Abstraction Refinement

The main technical challenge for model checking, especially of larger systems, is the *state-space explosion problem* [CGJ+03] It is the reason why algorithms are not applied to all-embracing models of the systems, but on simplified—abstract—models. The abstraction causes a loss of information incurred by simplifying the model. Hence, verifying the abstract model potentially leads to wrong results. These artificial specification violations are called *spurious counterexamples* [CGJ+03]. To thwart these potentially wrong results a technique called *Counterexample-Guided Abstraction Refinement* (CEGAR) has been proposed [CGJ+00; CGJ+03]. Information in the following paragraph is taken from [CGJ+03].

The technique starts with a small coarse model of the system to be verified and automatically gains information from spurious counterexamples in order to compute increasingly precise abstract models of the system. Each abstract model is an over-approximation of the original model and thus of the original system. A specification that holds in the abstract model will also hold in the concrete model. Nevertheless, a specification violation in the abstract model can either be caused by missing information, that means, it is spurious, or it can be a real violation. By refining the abstraction a spurious counterexample is eliminated. These increasingly precise abstract models are created until the specification is either proved or disproved by a counterexample.

CEGAR consists of several compontents: a *state-space exploration algorithm*, that is, an algorithm that computes the abstract model, for example, the CPA algorithm (see Section 2.3.2); a *precision* that determines the current level of abstraction; a *feasibility check* that decides on the feasibility of a found path and a *refinement* procedure, which is used to refine the precision of the abstract model [BLW15b]. The algorithm is depicted in Algorithm 2.2.

The state-space exploration algorithm explores the reachable abstract state space according to the current precision. The precision initially is coarse or even empty. The algorithm returns the verdict *true* if the state space was fully explored and no specification violation was found. This result lets the CEGAR algorithm terminate and it reports the *true* verdict. A found specification violation means that there is a path through the program that leads to the error location. The feasibility check decides whether this path is infeasible. For a feasible path the

**Algorithm 2.2:** CEGAR($\mathbb{D}, e_{\text{INIT}}, \pi_{\text{INIT}}$), for CPAs, taken from [BDW18]

**Input:** a composite CPA $\mathbb{D}$ composed of a location CPA $\mathbb{L}$, a ARG CPA $\mathbb{A}$, and possibly other CPAs;
for which $E$ denotes the set of abstract states and $\Pi$ the set of precisions;
with the additional operators fcover and refine;
and an initial abstract state $e_{\text{INIT}} = (l_{\text{INIT}}, \cdots) \in E$
with initial precision $\pi_{\text{INIT}} \in \Pi$

**Output:** *false* if $l_{\text{ERR}}$ is reachable, *true* otherwise

**Variables:** two sets reached and waitlist of elements of $E \times \Pi$ and a function $\text{abort}_{\text{ERR}} : E \to \mathbb{B}$

```
reached := {(e_INIT, π_INIT)};
waitlist := {(e_INIT, π_INIT)};
// abort_ERR returns true at l_ERR.
abort_ERR := {(l, ···) ↦ (l = l_ERR)};
while true do
    (reached, waitlist) := CPA++(𝔻, reached, waitlist, abort_ERR);
    if ∃((l_ERR, ···), ·) ∈ reached then
        (reached, waitlist) := refine(reached, waitlist);
        if ∃((l_ERR, ···), ·) ∈ reached then
            // refine has detected a feasible error path
            return false;
    else
        return true;
```

program is unsafe and the analysis terminates with a *false* verdict. In case of an infeasible error path the refinement procedure refines the precision in order to remove the infeasible error path from the abstract states space. Afterwards, the algorithm restarts and repeats until either a concrete error path is found or the abstract model is proven safe. The latter means that also the program is safe due to the over-approximating character of the analysis [BLW15b; BDW18].

## 2.5. Precision Adjustment

We call the information that determines the level of abstraction *precision* of the abstract states. We denote the set of possible precisions by $\Pi$ and use $\pi \in \Pi$ to denote an element thereof. We define a *program precision* as a mapping $L \to \Pi$ from the program locations $L$ to the set of precisions in order to support different precisions at different program locations.

The coarsest precision is called *empty precision*; it defines that all information is abstracted. Two precisions can be united by an *union*

operation. It is defined in an intuitive way; for example, the union of two predicate precisions, that means, the precisions are a set of predicates over program variables, is defined by the union of the predicate sets [BLN+13].

The process of applying a new precision to an abstract state is called *precision adjustment* and is done by the *precision adjustent operator* prec in the CPA algorithm (see Algorithm 2.1).

## 2.6. Craig Interpolation for Predicate Abstraction Refinement

A widely used technique in (software) model checking is predicate abstraction [GS97]. A set $\mathcal{P}$ of quantifier-free first-order predicates is used to model the program variables. In such an analysis, the precision $\pi$ is a set of predicates from $\mathcal{P}$.

By using a SMT solver it is possible to reason over the predicates. To refine such an abstract predicate-based model it is common to use *Craig interpolation* [Cra57] in order to discover information that allows the elimination of an infeasible error path. This information is called *interpolant*. Its structure is defined by Craig's interpolation theorem (see Theorem 1).

**Theorem 1 (Craig's interpolation theorem [Cra57; BL13]):** *Let $\varphi^-$, $\varphi^+$ be a pair of formulæ, such that $\varphi^- \wedge \varphi^+$ is unsatisfiable. Then there exists a formula $\psi$ (called* Craig interpolant*) that fulfils:*

*(i) the implication $\varphi^- \Rightarrow \psi$ holds,*

*(ii) the conjunction $\psi \wedge \varphi^+$ is unsatisfiable, and*

*(iii) $\psi$ only contains symbols that occur in both $\varphi^-$ and $\varphi^+$.*

The existence of interpolants is guaranteed for many common SMT theories; they can be computed by off-the-shelf SMT solvers [BDW18].

In traditional predicate abstraction refinement the atoms of the interpolants are extracted as predicates. From these predicates, the refinement procedure creates a new precision $\pi$ that then can be used to refine the abstract model.

# 3

# Provident Abstraction Refinement

We now present our provident abstraction refinement technique. Our presentation begins with the concept of a provident refinement. We then introduce our modified CEGAR-based algorithm that allows us to rank, select, delay, and combine possible refinements.

## 3.1. A Provident Strategy to Gain Information on Possible Refinements

Extracting good precisions, that is, precisions that lead to a fast convergence of the algorithm in as few steps as possible, from infeasible error paths is key to the CEGAR technique [BLW15b]. The selection of a refinement significantly influences the quality of the precision and the effectiveness of the analysis [BLW15c]. In existing approaches, that are based on lazy abstraction [HJM+02], the exploration of the state space stops after the first error location was found. For this location a refinement is performed. The quality of the retrieved precision lies only in the interpolation engine because the used predicate abstraction in CPAchecker relies on Craig interpolation. This is due to the fact that there might be several reasons for infeasibility in one path. Some of these reasons might be easier to track and can be more beneficial for the further analysis progress [BLW15c; BLW15b]. The interpolation engine, however, cannot decide which interpolants are best for the analysis progress because it has no access to this information. Furthermore, for an arbitrary interpolation problem the result cannot be controlled from the outside. This all can lead to divergence of the analysis [BLW15b].

Many CEGAR-based refinement strategies are based on the principles of lazy abstraction. Lazy abstraction [HJM+02] is a technique that allows to refine only parts of the abstract state space and keep the rest of it. This has the advantage that possibly expensive explorations of large parts of the reachable states does not have to be done over and over again. In order to achieve this, a refinement strategy based on lazy abstraction will pause the state-space exploration as soon as the first error location is found. If the path to this error location is feasible the analysis can terminate. Otherwise, there is some abstract state in which the abstract counterexample has an concrete counterpart; this state is called *pivot state*. In lieu of building an entire new abstract model, the current abstract model will be refined beginning from the pivot state.

The promise of this is that some parts of the abstract model are not relevant for the verification of the desired property, hence there is no need to re-explore them.

While refinement strategies based on lazy abstraction have proved to be prosperous, our approach does not follow this principle. In case an error location is found our state-space exploration algorithm shall not pause. We continue until the full reachable state space is explored. From this set of reachable abstract states we extract those that are error locations. Obviously, there can be more than one such error location. In the following, we can calculate a provident refinement for each such error location. A *provident refinement* is a refinement that does not do any changes to the set of reachable abstract states. Instead, we only want to extract the precision information. To do so, our framework reconstructs the error path from the abstract state's space and calculates interpolants—similar to the previously described approach. From these interpolants, we calculate the precision increment, that is, the new precision determined by the interpolants. The result of a provident refinement for one error location is a pair $(e, \pi)$ from the relation $E \times \Pi$ of abstract states and precisions.

In contrast to lazy abstraction, where a sequence of interpolants is corresponding to a sequence of program locations and thus also the precision, we break this relation and allow the precision to be applied to any state with any scoping. That is, a precision can not only be applied to a specific state and is only relevant for this state, but it can also be applied to any state being valid globally. This allows our framework to, for example, join the precisions of all found error paths into one precision containing all information. The resulting precision can then be applied to the root node of the ARG with a global scoping, that is, the precision is valid for this node and all its children, that is, the full space of reachable abstract states. After this is done, we restart the exploration of the state space. Similar to lazy abstraction, the framework provides an early exit in case a feasible counterexample is found.

In order to determine how this resulting precision shall look, our framework defines two operators, namely rank and select, which we will render more precisely in Section 3.3 and Section 3.4, respectively. Both operators influence the resulting precision; rank influences the order in which provident refinements are calculated for target locations while select provides the ability to determine the precision increment that will be applied to the abstract state space afterwards.

## 3.2. The PCEGAR Algorithm

We present the *Provident Counterexample-Guided Abstraction Refinement* (PCEGAR) algorithm in this section. It is based on a classical CEGAR algorithm similar to the one described in Section 2.4. The algorithm is listed in Algorithm 3.1. Technically, it is a semi-algorithm because

**Algorithm 3.1:** PCEGAR($\mathbb{D}, e_{\text{INIT}}, \pi_{\text{INIT}}$) for CPAs

**Input:** a composite CPA $\mathbb{D}$ composed of a location CPA $\mathbb{L}$, a ARG CPA $\mathbb{A}$, and possibly other CPAs;
for which $E$ denotes the set of abstract states and $\Pi$ the set of precisions;
with the additional operators extract_targets, rank, provident_refine, select, and apply;
and an initial abstract state $e_{\text{INIT}} = (l_{\text{INIT}}, \cdots) \in E$
with initial precision $\pi_{\text{INIT}} \in \Pi$

**Output:** *false* if $l_{\text{ERR}}$ is reachable, *true* otherwise

**Variables:** three sets reached, waitlist and candidates of elements of $E \times \Pi$ and a function $\text{abort}_{\text{ERR}} : E \to \mathbb{B}$

```
reached := {(e_INIT, π_INIT)};
waitlist := {(e_INIT, π_INIT)};
// abort_ERR returns false to explore the full state space
abort_ERR := {(l, ···) ↦ false};
candidates := ∅;
while true do
    (reached, waitlist) := CPA++(𝔻, reached, waitlist, abort_ERR);
    if ∃((l_ERR, ···), ·) ∈ reached then
        targets := extract_targets (reached);
        targets := rank (targets);
        candidates := ⋃_{t∈targets} provident_refine(t);
        s := select (candidates);
        (reached, waitlist) := apply(reached, waitlist, s);
        if ∃((l_ERR, ···), ·) ∈ reached then
            // refine has detected a feasible error path
            return false;
    else
        return true;
```

termination cannot be guaranteed; this can be the case, for example, if we choose PCEGAR's operators in a way, it does not remove a spurious counterexample. Still, we omit the prefix semi in our presentation for simplicity, and call PCEGAR an algorithm in the following.

If we omit the additional operators extract_targets, rank, provident_refine, and select for now, the algorithm is identical to the CEGAR algorithm presented in Algorithm 2.2. The only difference is that we need to replace PCEGAR's apply operator by CEGAR's refine operator.

In the following, we assume the operator rank to be an identity relation, and the select operator to always select the first element given in its parameter list. This is for simplicity of the algorithm's presentation. We will discuss the rank operator in Section 3.3 and the select operator in Section 3.4.

First of all, PCEGAR is a refinement algorithm, that means, it only refines the abstract state space explored by an external state-space exploration algorithm. One such state-space exploration algorithm is the CPA algorithm (see Section 2.3.2 for details). The exploration algorithm is meant to explore a set of reachable abstract states, called the reached set, and denoted by reached. Among the states in the reached set, there can be states that symbolise a violation of a property, which we call *target states* in the following. If there are no such states and the state space was fully explored, the program is considered to be safe. This leads to a termination of the refinement algorithm; hence, it returns the verdict *true*.

In case there exists at least one such target state, it is necessary to determine, whether it is a real violation of the property, or if it exists just because of a too coarse abstract model. For that, our refinement algorithm extracts all such target states using the operator extract_targets, which simply filters the set of reachable abstract states for those that fulfil the target-state criteria. The resulting list is then given to the ranking operator rank.

For each such target state there exists a path through the ARG that connects the target state with the entry point of the program, which is represented by the root node of the ARG. Let $E$ be the set of all reachable abstract states. Also, let $\langle e_{\text{INIT}}, e_1, e_2, \dots, e_{\text{ERR}} \rangle$ with $e_{\text{INIT}}, e_1, e_2, \dots, e_{\text{ERR}} \in E$ be an infeasible path in the set of reached states. Then there exists an abstract state $e_m \in E$ with $e_m \in \langle e_{\text{INIT}}, e_1, e_2, \dots, e_{\text{ERR}} \rangle$, which causes the infeasibility of this path in the program but not in the abstract model. Using Craig interpolation, predicates can be generated fully automatically from a proof of unsatisfiability for the formula representing such a spurious counterexample [HJM+04]. Because our analysis is based on predicate abstraction, where a precision $\pi$ is a set of predicates [BLN+13], we can use these predicates from Craig interpolation as our new precision. Finally, there is a pair $(e, \pi)$ from the relation $E \times \Pi$ for each such provident refinement, where $e$ is the pivot state as used in lazy abstraction. Note that this does not limit our technique to analyses based on predicate abstraction. It was shown, for example by Beyer and Löwe [BL13], that also other analysis types, such as explicit-value program analysis, benefit from abstraction, CEGAR, and interpolation.

The list of such pairs is then given to the selection operator select. The operator returns a single pair $(e, \pi)$, which describes the actual refinement, that is, the new precision that shall be added to the abstract model in the next step. Finally, the operator apply adds the new precision to the relevant abstract states like a standard CEGAR-based refinement strategy does.

Afterwards, it is checked, whether the target state $e_{\text{ERR}}$ is still reachable. As we considered the error path to be infeasible, $e_{\text{ERR}}$ cannot be reachable any more. Hence, the algorithm continues to explore the

state space again. If the error path was feasible in the first place, the algorithm would have stopped here by returning the verdict *false*.

## 3.3. The Ranking Operator

The target ranking operator rank is the first of two operators defined by our framework. As stated before, it is used to determine the order in which provident refinements are calculated for the found target locations. Formally, we define the operator as given by Definition 1.

**Definition 1 (The Operator rank):** *Let $I$ be the non-empty list of abstract target states in random order (as determined by the state space exploration strategy), that is, each element $i$ of $I$ is an element of the set of abstract states $E$ and is a target state. Let further $O$ be non-empty sub-list of $I$, ordered by an order relation $\preceq$.*

*We then define the operator* $\mathsf{rank}_{\preceq}$ *as a surjection between the lists $I$ and $O$:*

$$\mathsf{rank}_{\preceq} : I \to O$$

This definition leads to two important remarks: we first note, that neither $I$ nor $O$ shall be empty. The simple reason is that the provident refinement algorithm uses $O$ as input for its calculations; for an empty list it is not possible to calculate anything, thus, there would not be a single counterexample removal during the refinement procedure, which can lead to non-termination of the PCEGAR algorithm. In case $I$ was empty, the algorithm would already have stopped earlier, because there was no target state in the reached set anyway. Second, if we refer to the ranking operator for a specific order $\preceq$, we denote it by $\mathsf{rank}_{\preceq}$; if we refer to the ranking operator in general, where we are not interested in the order, we omit the order and denote the operator only by rank.

The order $\preceq$ is necessary to determine the position of an element of $I$ in the output list $O$. It can, for example, use the smallest number of abstract states that are on a path between two abstract states in the ARG. The order is used for sorting the elements of $I$.

## 3.4. Strategies to Select a Refinement

Crucial to the provident refinement framework is the selection operator that takes the calculated provident refinements and combines them to a compound refinement that will then be applied to the abstract state's space. Before we can define the operator, we need some notation convention:

**Definition 2 (Notation):** *By $\langle E \times \Pi \rangle$ we denote a list of pairs $(e_i, \pi_i)$ of abstract states and precisions. By $\langle (e_1, \pi_1), \ldots, (e_n, \pi_n) \rangle$ we refer to a list of length $n \in \mathbb{N} \setminus \{0\}$ of such pairs of $\langle E \times \Pi \rangle$. Core of this list is that it is an ordered structure in contrast to, for example, a set.*

Using this notation, we can give a formal definition of the selection operator in Definition 3.

**Definition 3 (The Operator select):** *Let $E$ be the set of abstract states and $\Pi$ be the set of precisions. We define the operator* select *as*

$$\text{select} : \langle E \times \Pi \rangle \rightarrow E \times \Pi$$

The operator gets as input a list of pairs of abstract states and precisions and returns one pair of abstract state and precision. It is important to notice that neither input nor output value of the operator shall be empty.

The selection operator select is the core of the PCEGAR framework. It gives the possibility to combine, select, or delay refinements, by emitting an appropriate result. When more than one pair from the input set is used for calculating the output of the operator, we have the problem of how to combine the information about the abstract states. In this case, the operator uses the ARG's root as the abstract state in the resulting pair. There is only one root node in the ARG, which is effectively an abstract reachability tree, because there can be only one abstract state that corresponds to the initial program location. This is sound because we apply the precision with a global scoping afterwards, that is, the information gets propagated to every abstract state in the reached set anyway.

Joining Provident Refinements Obviously, selecting only one pair from the set of input pairs is trivial. Thus, we now focus on the combination of at least two of them. As stated before, in this case the abstract state $e_r$ in the resulting pair will always be the ARG's root node. The resulting precision $\pi_r$ can simply be calculated by joining all precisions, that is, $\pi_r = \bigcup_{i \in \{1,\ldots,n\}} \pi_i$ for a join of $n$ precisions. This leads to the result $(e_{\text{INIT}}, \pi_r)$ for such a join.

Delaying Provident Refinements As stated before, it can be reasonable to prefer some refinements over others. For example, refining for a loop counter can lead to expensive loop unrollings [BLW15b]; this can be undesirable. Therefore, delaying such refinements can improve the performance of the analysis. It might, however, still be necessary to execute them at some point and, for example, unroll loops, but often this can be avoided.

Delaying a refinement by the select operator means that it simply will not incorporate the refinement's precision information into the operator's result. Depending on the progress of the algorithm the delayed refinement will be re-explored at some point or does not appear at all if it is not necessary for the algorihm's result. The latter is the case, if paths to the target state corresponding to such a refinement, have been ruled out by other refinements before.

Note on Implementation Note that the implementation of the selection operator differs from this formal definition, such that it does not only get a list of tuples of $E \times \Pi$. It furthermore gets additional information generated by the provident refinement, such as the related ARG path, as well as information, whether the found counterexample is spurious. This is an optimisation to guarantee early termination of the PCEGAR algorithm and avoid expensive recalculations. Consider a list of provident refinements where all except one counterexamples are spurious. One counterexample is a real specification violation. A selection operator that only takes a subset of the provided counterexamples into account, might miss the feasible counterexample and cause further refinements for the spurious ones. Although, the feasible counterexample will be found at some point and the analysis will terminate with the expected result, it is possible that this will occur much later than necessary. In case the set of counterexamples contains a feasible one, the selector always selects the feasible counterexample, as this will lead to an early termination. This is sound, as the feasible counterexample would be found at some point anyway.

# 4

# Evaluation

We provide an empirical study on our provident multi-path refinement framework in this chapter. We start by our research questions in Section 4.1. Thereafter, we define certain concrete operator implementations in Section 4.2 and present our hypotheses we want to check during the evaluation in Section 4.3. Furthermore, we present our case study in Section 4.4 and explain the environment we used in Section 4.5. In Section 4.6 we present the results and discuss them in Section 4.7. Finally, we conclude this chapter with a discussion of the threats to validity in Section 4.8.

## 4.1. Research Questions

We now define questions that guide our evaluation. We evaluate each question by using a set of verification tasks. A *verification task* is a pair of a program and one or many properties. Some of our questions target the *performance* of the verification tasks in terms of efficiency and effectiveness. A configuration is considered to be more *effective* than another one if it is able to provide verification result for more tasks than the other analysis. It is considered to be more *efficient* if it can solve given verification tasks by consuming less resources in terms of CPU time.

Research has shown that even on a single path to an error location there can be many reasons for the infeasibility of this abstract path. Considering the information about every such reason for infeasibility and selecting the most appropriate in the specific situation is targeted, for example, with sliced path prefixes [BLW15c] or refinement selection [BLW15b]. These approaches consider only one path. It seems probable that by incorporating the information of more than one path into the selection process of a refinement, the results can even be improved.

**Research Question 1 (RQ1):** *Is there a multi-path refinement strategy that performs better than a single-path refinement strategy in terms of efficiency and effectiveness?*

Not only do we expect that incorporating more information into the refinement strategy can influence the performance of a verification

run, but it is also expected that the structure of a verification task has an influence. Such an influence can be determined by the program, obviously. A program consisting of neither loops nor recursion, for example, cannot cause expensive loop unrolling during the refinement. The same holds for the property. Properties with a scope that is very local to only a few lines of code or a few functions cause different behaviour, than properties with a wide scope over large parts of a program [Dah17].

**Research Question 2 (RQ2):** *Are there specific structures (characteristics) in the verification tasks, for that multi-path refinement has considerably different performance characteristics than single-path refinement?*

Furthermore, it is expected that by incorporating domain knowledge into an analysis, we can profit from this knowledge. By domain knowledge, we think, for example, of characteristics like many global variables or the intense usage of pointer arithmetics. Also the position of target states in the ARG is an interesting characteristics. Using this information when building an appropriate strategy influences its performance.

**Research Question 3 (RQ3):** *Does incorporating domain knowledge (characteristics of the verification task) into a multi-path refinement strategy pay off in terms of efficiency and effectiveness compared to a single-path refinement strategy?*

## 4.2. Operators for PCEGAR

Let us now define some operator implementations we refer to in the following sections. We give implementations for the rank and select operators in this section and assign symbols to identify the operator implementations during the presentation. Formal definition and requirements on these operators can be found in Sections 3.3 and 3.4.

### *4.2.1. Ranking Operator Implementations*

We define some concrete examples of target ranking operators. This list does not make a claim to be complete. We only present those operators, we use in the presentation of our results. The most straightforward ranker, which we call *identity target ranker*, is an identity operation. It simply emits the given input list as output without touching it. We refer to this identity target ranker by the symbol $\mathcal{R}_{\mathsf{ID}}$. A second possible ranker always takes the first element from the operator's input list and returns a list with this single element; we refer to it as $\mathcal{R}_{\mathsf{F}}$.

Finally, two further target rankers do effectively change the elements of their input lists. Each abstract state has a property *state level* that determines its distance to the ARG's root. Both rankers order the elements of their input lists based on this state level. The first (referred to as $\mathcal{R}_{\mathsf{T}}$)

**Table 4.1.** Overview over the introduced ranking operators

| Symbol | Description |
|---|---|
| $\mathcal{R}_{\mathsf{ID}}$ | The identity operation. Does no ranking at all. The input list will be returned without any changes. |
| $\mathcal{R}_{\mathsf{F}}$ | Does no ranking at all. The first element of the input list will be returned as the resulting one-element list. |
| $\mathcal{R}_{\mathsf{T}}$ | Takes the abstract states in the input list and sorts them ascending with regard to their state level. The result contains the same states as the input but in a different order. |
| $\mathcal{R}_{\mathsf{B}}$ | Takes the abstract states in the input list and sorts them descending with regard to their state level. The result contains the same states as the input but in a different order. |

does it in ascending order; the second (referred to as $\mathcal{R}_{\mathsf{B}}$) does it in descending order of the state level. Table 4.1 provides a comprehensive overview over these four target ranking operators.

### 4.2.2. *Selection Operator Implementations*

In addition to the ranking operator, the PCEGAR framework defines an operator to select a refinement that gets applied; the selection operator select. We define some possible implementations of this operator without claiming completeness of this list. The presented implementations are only those that are used during the evaluation.

First, we introduce a simple select first operator, which we refer to as $\mathcal{S}_{\mathsf{F}}$. It selects the first element from its input list and returns it without further ado. Formally, for a list of pairs of abstract state and precision, denoted by $\langle (e_1, \pi_1), \dots, (e_n, \pi_n) \rangle$, the operator is defined as

$$\mathcal{S}_{\mathsf{F}} : \langle (e_1, \pi_1), \dots, (e_n, \pi_n) \rangle \mapsto (e_1, \pi_1)$$

The second operator we introduce, is a selector that joins together all precisions from its input list of pairs of abstract states and precisions. It uses the ARG's root node as the abstract state in the resulting pair. We denote this all-join selection operator by $\mathcal{S}_{\bigcup}$. Formally, for a list of pairs of abstract state and precision, denoted by $\langle (e_1, \pi_1), \dots, (e_n, \pi_n) \rangle$, and the ARG root state $e_{\text{INIT}} \in E$, we define the operator as

$$\mathcal{S}_{\bigcup} : \langle (e_1, \pi_1), \dots, (e_n, \pi_n) \rangle \mapsto \left( e_{\text{INIT}}, \bigcup_{i \in \{1,\dots,n\}} \pi_i \right)$$

Lastly, the third operator takes the first up to $k$ elements from the input list and joins them together. Note that for input lists with a size $m$ less than $k$, that is, $m < k$, only the $m$ elements are joined. As before, the ARG's root node is used as the abstract state in the resulting pair. We denote this operator by $\mathcal{S}_{\leq \mathsf{k}}$, where $k \in \mathbb{N} \setminus \{0\}$ is the number of elements

**Table 4.2.** Overview of the introduced selection operators

| Symbol | Description |
|---|---|
| $\mathcal{S}_{\mathsf{F}}$ | Selects only the one refinement from the list of all possible refinements that is on the first position of the input. |
| $\mathcal{S}_{\cup}$ | Takes all possible refinements into account and joins their precisions to a new precision. |
| $\mathcal{S}_{\leq \mathsf{k}}$ | Takes the first $k$ elements from the list of all possible refinements and joins their precisions to a new precision. The value $k \in \mathbb{N} \setminus \{0\}$ can be configured. |

that shall be joined. We can formally define the operator for a list of pairs of abstract state and precision, denoted by $\langle (e_1, \pi_1), \ldots, (e_n, \pi_n) \rangle$, a number of elements $k \in \mathbb{N} \setminus \{0\}$, and the ARG root state $e_{\text{INIT}} \in E$, as

$$\mathcal{S}_{\leq \mathsf{k}} : \langle (e_1, \pi_1), \ldots, (e_n, \pi_n) \rangle \mapsto \begin{cases} \left( e_{\text{INIT}}, \bigcup_{i \in \{1,\ldots,k\}} \pi_i \right) & \text{if } k \leq n \\ \left( e_{\text{INIT}}, \bigcup_{i \in \{1,\ldots,n\}} \pi_i \right) & \text{if } k > n \end{cases}$$

Table 4.2 gives a comprehensive overview over the presented and implemented selection operators.

### 4.2.3. *Particular Operator Combinations*

In principle, it is possible to freely combine all ranking with all selection operators. Obviously, not all combinations have different behaviours from each other. For example, consider a combination of $\mathcal{R}_{\mathsf{ID}}$ and $\mathcal{S}_{\mathsf{F}}$; this is equal to $\mathcal{R}_{\mathsf{F}}$ and $\mathcal{S}_{\mathsf{F}}$ because the selector takes in both cases only the first element of its input list. The result is equivalent, independent of the number of elements the ranking operator's output list consists of.

Two combinations are outstanding for the following presentation. We call a combination of $\mathcal{R}_{\mathsf{F}}$ and $\mathcal{S}_{\mathsf{F}}$ a *single-path refinement strategy*. We further call the combination of $\mathcal{R}_{\mathsf{ID}}$ and $\mathcal{S}_{\cup}$ the *multi-path refinement strategy*.

## 4.3. Hypotheses

We now introduce the hypotheses that guide the evaluation and give a detailed insight into the relevant configuration settings of our framework. Each experiment consists of two verification run sets, each consisting of a number of verification tasks. Our run set consists only of those tasks that caused at least one refinement because we are particularly interested in refinement strategies. Hence, we omit tasks without refinements from the experimental evaluation. We furthermore omit verification tasks that caused a timeouts or other errors due to the fact that these are incomplete verification runs.

All configurations of our framework that are presented in the following are derived from the same base configuration. They only differ in the selection of the ranking operator (see Section 3.3) and the selection operator (see Section 3.4). By *single-path refinement strategy*, we denote a configuration that always takes the first found error path for precision calculation (ensured by operator $\mathcal{R}_{\mathsf{F}}$) and only applies this single precision increment (ensured by operator $\mathcal{S}_{\mathsf{F}}$).

### *4.3.1. Multi-Path versus Single-Path Strategy*

In the first group of hypotheses we are particularly interested in the performance behaviour of a multi-path refinement strategy compared to the single-path refinement strategy as a base line. Therefore, we configure the multi-path refinement strategy in a way that it incorporates all found information in each refinement step. This can be achieved by the all-join selection operator $\mathcal{S}_{\bigcup}$. As the operator simply joins all found precisions to a final precision that is used for refining the state space, a ranking is not necessary at all. To minimise the cost of the ranking operator we use the identity operator $\mathcal{R}_{\mathsf{ID}}$, which does not affect the found target states and their order at all.

For the evaluation of the first research question, we use verification tasks, where a Linux kernel module is combined with all 14 safety properties, that is, a multi-property verification [ABM+16]. Each property has a specific structure, that is, it is relevant for different scopes in the program. Some properties are only relevant for a few lines or functions of the program, others are relevant for large parts of the program [Dah17]. By virtue of the multi-property verification, we try to cancel out the influence of a single property's structure.

We anticipate that our multi-path refinement strategy will decrease the CPU time that is needed for the verification compared to the single-path refinement strategy. By *(analysis) CPU time*, we refer to the time that is needed only for the verification process itself. This excludes steps like parsing of the program and the specifications, building the framework's internal data structures, and so forth. It is valid to omit these steps, as they occur both for multi-path and single-path refinement; they do not influence the refinement strategy at all, because these steps happen either before or after the refinement strategy is executed.

**Hypothesis 1 (H1):** *There exists a multi-path refinement strategy that, for a set of verification tasks, in general consumes less CPU time, than a single-path refinement strategy.*

Not only the efficiency of a configuration is a relevant metrics, but also its effectiveness. Combining the insights on both, efficiency and effectiveness, allows us to reason about the analysis' performance. We again use the multi-property verification tasks and the same operator configurations like before; furthermore, all other settings are the same,

too. Our presumption is that the multi-path refinement strategy can solve more verification tasks correctly, than the single-path strategy does.

**Hypothesis 2 (H2):** *There exists a multi-path refinement strategy that, for a set of verification tasks, in general, can solve more verification tasks, than a single-path refinement strategy.*

### 4.3.2. Influences of the Verification Tasks

The behaviour of a verification tool is also determined by the verification tasks. In particular, the performance can differ significantly by verifying many properties in one run compared to verifying each independently [ABM+16].

We also expect such behaviour for our multi-path refinement strategy, when comparing it to a single-path strategy. Hence, we are not interested in the speedup we can gain from verifying the properties in one run compared to verifying them seperately. Instead, we take a look at how different properties affect the analysis. That is, we want to know, whether it is possible to gain a speedup from a multi-path refinement strategy independently of the used property.

In order to do our experiments, we again use the multi-path strategy, consisting of the operators $\mathcal{R}_{\mathsf{ID}}$ and $\mathcal{S}_{\cup}$, and compare it to the single-path strategy, consisting of the operators $\mathcal{R}_{\mathsf{F}}$ and $\mathcal{S}_{\mathsf{F}}$. Our verification tasks for this question consist of the 250 Linux kernel modules each combined with one of the 14 properties, which leads to 3 500 verification tasks. Again, we anticipate that our multi-path strategy consumes less CPU time for the analysis, than the single-path strategy for all verification tasks.

**Hypothesis 3 (H3):** *For verification tasks consisting of different properties our multi-path refinement strategy always consumes less CPU time than our single-path refinement strategy.*

### 4.3.3. Incorporating Domain Knowledge

The incorporation of domain knowledge in the analysis is expected to improve its performance. We expect that preferring target states with a small distance to the ARG's root[1] can safe refinements, compared to preferring target states with large distance to the root. This expectation is based on the assumption that a refinement for a target state near to the root rules out paths through the ARG that lead to further target states, because these paths branch from our refined path.

In the experiment we use the number of refinements from our single-path and our multi-path strategy as base lines. The ranking operators $\mathcal{R}_{\mathsf{T}}$ and $\mathcal{R}_{\mathsf{B}}$ ensure that we have the target states sorted according

[1] By unrolling loops, the ARG is transferred into a tree.

**Table 4.3.** An overview over independent, dependent, and controlled variables per hypothesis

| Hyp. | Variables | | |
|---|---|---|---|
| | Independent | Dependent | Controlled |
| H1 | ranking and selection operators | CPU time | verification tasks, framework configuration |
| H2 | ranking and selection operators | number of solved tasks | verification tasks, framework configuration |
| H3 | verification tasks | CPU time | ranking and selection operators, framework configuration |
| H4 | ranking and selection operators | number of refinements | verification tasks, framework configuration |

to their state level. The former sorts the target states in ascending order starting with the target state with the smallest distance from the root, while the latter sorts the states in descending order. Using the all-join selection operator $\mathcal{S}_{\cup}$ is not reasonable here, because it does not incorporate the order of the target states when joining all available precisions. The different order of predicates given to the SMT solver from this approach cannot be controlled, because the solver can reorder the clauses for internal optimisations anyway. Hence, we join only fractions of the possible precision information using the operator $\mathcal{S}_{\leq k}$ with $k \in \{2, 4, 8\}$. Different values for $k$ enable us to get further insights on how the number of necessary refinements changes, when incorporating the information of a different number of provident refinements.

**Hypothesis 4 (H4):** *By preferring target states having a small distance to the ARG's root, a multi-path refinement strategy saves refinements compared to a multi-path refinement strategy that does not prefer any states.*

### *4.3.4. Variables*

For each research question, we identify a number of variables that determine the experiments. We categorise the variables in independent, dependent, and controlled variables. Table 4.3 provides an overview over the variables for each hypothesis. Independent are those variables we deliberately change throughout our experiments. Changes on independent variables affects the dependent variables, which are those variables we use for our measurements. The controlled variables also influece the results; we keep them fixed such that they do not influence the results in incontrollable ways.

The framework's overall configuration is determined by options like its abstraction strategy, the used abstract domains, the SMT solver, and others. It is not changed throughout the experiments, hence, we consider it to be a controlled variable.

We designed the experiments based on the hypotheses and these variables to answer our research questions.

## 4.4. Case Studies for Evaluation

For our evaluation we used a set of 250 modules from the Linux kernel version 4.0-rc1, which were also used in previous work [ABM+16]. The benchmark files were created using the *Linux Driver Verification* (LDV)[2] toolkit [KMP+09]. The LDV toolkit also enriches the kernel modules with an environment model of the Linux kernel. The pre-processed Linux kernel modules are licensed under GNU GPL 2.0.

Together with the kernel modules, we use a set of 14 safety properties (adjusted versions from the ones used in [ABM+16]) that are relevant for the Linux kernel and describe the proper usage of its API. We provide a detailed description of each property in Table 4.4.

From the 250 Linux kernel modules we only consider those for our analysis where at least one refinement is necessary. This is due to the fact that we compare refinement strategies and consider the performance behaviour to be independent of the strategy if no refinement is necessary at all. Due to this restriction we have a different number of verification tasks for the evaluation of each hypothesis. Each benchmark task is a combination of one Linux kernel module and one or all 14 properties.

## 4.5. Evaluation Environment

Our approach is implemented in a fork of CPAchecker; we use revision `e027976633` (GIT tag `provident-refinement-thesis`) from its GIT repository[3]. The fork is based on CPAchecker's version 1.6; an earlier version of the fork was used before [ABM+16]. All files that are necessary for reproducing the evaluation can be found on the supplementary web page[4].

All evaluation runs were performed on machines with two Intel® Xeon® E5-2650 v2 CPUs, with 16 processing units each, and a clock frequency of 3.40 GHz. Each machine is equipped with 135 GB RAM and runs Ubuntu 16.04 (64 bit edition) based on a Linux 4.4 kernel. The machines have the 64 bit *Java Virtual Machine* (JVM) OpenJDK 1.8.0_151 installed. We disabled Turbo Boost and Hyper Threading and, furthermore, have bound each CPU to its local memory banks to reduce possible effects of the NUMA shared memory architecture [MG11]. Our

[2] See http://linuxtesting.org/project/ldv/ for more information.
[3] See https://github.com/se-passau/pytheas
[4] See https://research.lukasczyk.me/provident-refinement

**Table 4.4.** Safety properties for the Linux kernel modules (taken from [ABM+16])

| Property | Description |
|---|---|
| 08_1a | Each module that was referenced with `module_get` must be released with `module_put` afterwards. |
| 10_1a | Each memory allocation that gets performed in the context of an interrupt must use the flag `GFP_ATOMIC`. |
| 32_1a | The same mutex must not be aquired or released twice in the same process. |
| 43_1a | Each memory allocation must use the flag `GFP_ATOMIC` if a spinlock is held. |
| 68_1a | All resources that were allocated with `usb_alloc_urb` must be released with `usb_free_urb`. |
| 68_1b | Each DMA-consistent buffer that was allocated with `usb_alloc_coherent` must be released by calling `usb_free_coherent`. |
| 77_1a | Each memory allocation in a code region with an active mutex must be performed with the flag `GLP_NOIO`. |
| 101_1a | All structs that were obtained with `blk_make_request` must get released by calling `blk_put_request` afterwards. |
| 106_1a | The modules `gadget`, `char`, and `class` that were registered with `usb_gadget_probe_driver`, `register_chrdev`, and `class_register` must be unregistered by calling `usb_gadget_unregister_driver`, `unregister_chrdev`, and `class_unregister` correspondingly in reverse order of the registration. |
| 118_1a | Reader-writer spinlocks must be used in the correct order. |
| 129_1a | An offset argument of a `find_bit` function must not be greater than the size of the corresponding array. |
| 132_1a | Each device that was allocated by `usb_get_dev` must get released with `usb_put_dev`. |
| 134_1a | The `probe` functions must return a non-zero value in case of a failed call to `register_netdev` or `usb_register`. |
| 147_1a | RCU pointer/list update operations must not be used inside RCU read-side critical sections. |

framework is built in the Java programming language. Because of the JVM and its *Just-in-Time* (JIT) compiler we force the JVM to compile most of the code already during its startup to mitigate the effects of its JIT compiler.

For our evaluation, we set a CPU time limit of 300 s and a memory limit of 15 GB. The Java heap of the JVM is set to 13 GB. Our analyses use MathSAT 5.3.11 [CGS+13] as SMT solver. We use BenchExec [BLW15a; BLW17] to achieve reliable benchmarking results.

For all runs we measured the amount of CPU time that is necessary for the analysis only. This means, we omit the time for setting up the analysis, parsing the verification task, and so forth, because they are not relevant for the performance measures we are interested in. In all tables and figures, we give the time in seconds and the memory consumption

in megabytes with three significant digits unless stated other. Counted numbers, like the number of algorithm iterations or the number of refinements, are given in full precision. For the conversion of bytes we use SI units, that is, 1 MB is composed of 1 000 kB.

## 4.6. Results

In the following we analyse the benchmark results in order to answer our research questions and test our hypotheses. We dropped all runs from the presentation, which did not pass with with verdict *true*. These are all runs that did not finish within the timeout of 300 s, runs where the framework crashed for whatever reason—parser errors, failures of the SMT solver etc. We also omit all results with a *false* verdict; this is because we assume the Linux kernel modules do not contain specification violations even though we cannot verify whether *false* verdicts are real violations or false positives.

Note that we add a line on the diagonal of scatter plots to get a better insight on the distribution of the results. Furthermore, dashed lines in scatter plots annotated with numbers like "×1.5" denote the speedup—in this case a speedup of 1.5.

### *4.6.1. Multi-Path versus Single-Path Strategy*

In the first research questions we are asking whether a multi-path refinement strategy can outperform a single-path refinement strategy. To evaluate this, we compare the results of both strategies on our Linux kernel modules with all 14 specifications.

**Efficiency** In Hypothesis 1, we formulated the expectation that the multi-path refinement strategy can benefit from the growth of information, which is incorporated into the refinement; we expect that it can solve given tasks in less CPU time. Figure 4.1 compares both strategies in a scatter plot. The abscissa gives the CPU time (in seconds) for the single-path refinement strategy; the strategy is determined by the operators $\mathcal{R}_{\mathsf{F}}$ and $\mathcal{S}_{\mathsf{F}}$. On the ordinate axis we give the CPU time (also in seconds) for the multi-path refinement strategy; this strategy is determined by the operators $\mathcal{R}_{\mathsf{ID}}$ and $\mathcal{S}_{\cup}$. As stated before, the line on the diagonal gives a hint on equal values; we furthermore include lines indicating speedups of factors 1.5 and 2, respectively. Each data point shows the CPU times for one Linux kernel module for both strategies. Data points on the diagonal indicate equal CPU times. Data points in the lower right triangle indicate that the CPU time for multi-path refinement was smaller than the CPU time for single-path refinement. The vice versa holds for data points in the upper left triangle.

Out of the 250 Linux kernel modules in the benchmark set, we consider only those that cause at least one refinement, result in a *true*

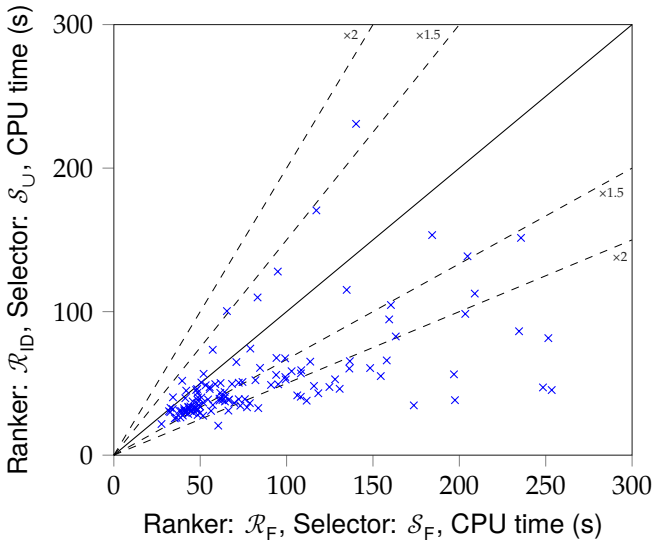


**Figure 4.1.** Comparison of the efficiency of multi-path versus single-path refinement strategy in terms of analysis CPU time. Marks in the right lower triangle are better for multi-path.

verdict, and where the verdict was the same for both the single-path and multi-path configurations. This restrictions result in a number of 128 verification tasks we consider for our contemplations. From Figure 4.1 we can see that most of the data points lie in the lower right triangle. This means, the multi-path strategy needs less CPU time than the single-path strategy to solve them. The mean speedup of the multi-path refinement strategy is 1.71; its median is 1.47. *Remark:* a speedup of 2 between a configuration $A$ and a configuration $B$ connotes that $A$ is twice as fast as $B$, that is, it consumes half the CPU time for a task.

For only 11 of the 128 tasks (8.59 %) we encounter a slow-down, that is, the single-path strategy is faster than the multi-path strategy. In the worst case, the single-path strategy is 1.65 times faster than the multi-path refinement strategy. The remaining 117 tasks get solved faster the multi-path strategy with a maximum speedup of 5.59.

Most of the verification tasks (56, that is 43.8 %) can be solved with a speedup between 1 and 1.50 by the multi-path strategy. For 34 verification tasks (26.6 %) the speedup is between 1.50 and 2; in 27 cases (21.1 %) the speedup is even larger than 2.

The refinement strategy has a significant impact on the performance of an analysis. Particularly, the number of refinements is crucial—it is assumed that an analysis benefits from saving refinements, hence will be faster. We can see from Table 4.5 that our single-path refinement strategy needs about five times as much executed refinements as the multi-path strategy. Note that the equal numbers for provident and executed refinement for the single-path strategy are expected because

**Table 4.5.** Numbers of provident and executed refinements for single-path and multi-path strategy

| Name | Min | Median | Average | Max | Sum |
|---|---|---|---|---|---|
| Single-path provident refinements | 1 | 5.00 | 6.10 | 29 | 781 |
| Single-path executed refinements | 1 | 5.00 | 6.10 | 29 | 781 |
| Multi-path provident refinements | 11 | 29.5 | 43.5 | 356 | 5 560 |
| Multi-path executed refinements | 1 | 1.00 | 1.26 | 6 | 161 |

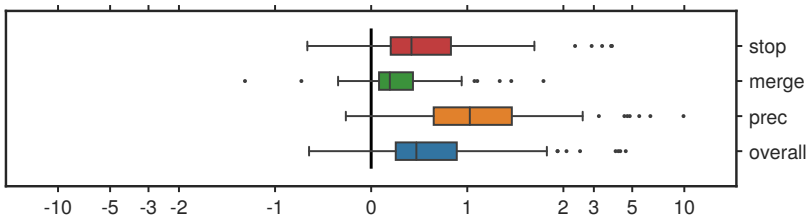


**Figure 4.2.** Comparing the speedups for different components and operators of the framework. Left of the middle bar is the single-path strategy, on the right side the multi-path strategy Note that a speedup of 1 in the figure is a speedup of 100 %, that is, execution time of the faster configuration is only half of the time the slower configuration takes.

every calculated provident refinement will be applied to the abstract state's set immediately.

We can see that we benefit from doing a multi-path refinement when only looking at the number of executed refinements. Nevertheless, is it also possible to identify an operator of the CPA framework that indicates the speedup? Thereto, we calculate the speedups of the operators stop, merge, prec, and the overall speedups. Figure 4.2 shows the results. Values left of the vertical bar in the middle of the plot indicate verification tasks where the single-path refinement strategy is faster than the multi-path refinement strategy; it is vice versa for data points right of the bar. For none of the operators we can see large speedups on most of the verification tasks. Only for the precision-adjustment operator prec we see a larger improvement. We expect this as the operator adjusts the precisions—a process that is done during refinement. Hence, if we save refinements, the operator will not be called that often, which results in the lower consumed time.

Overall we see an improvement in terms of consumed CPU time, where our multi-path refinement strategy outperforms the single-path refinement strategy.

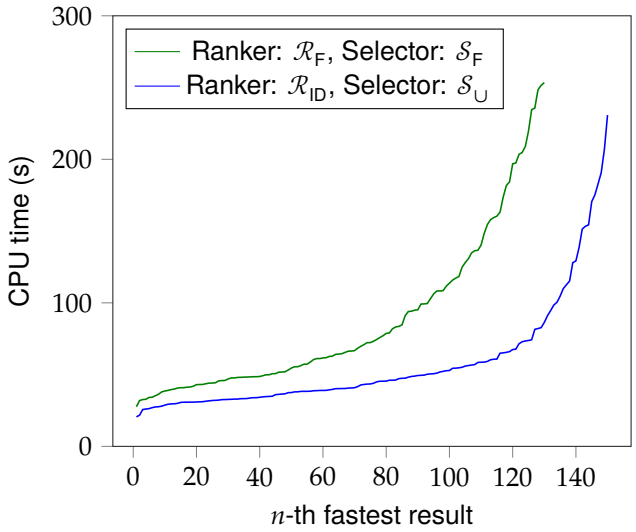


**Figure 4.3.** Comparison of the effectiveness of multi-path versus single-path refinement strategy in terms of analysis CPU time. Lower and more right data points are better.

**Summary (*Hypothesis 1*):** On average, the single-path refinement strategy needs 1.71 (median: 1.47) times the CPU time the multi-path refinement strategy does for all correctly solved verification tasks that cause at least one refinement, which are 128 of 250. In conclusion, we **accept Hypothesis 1**.

Effectiveness In Hypothesis 2, we formulated the expectation that the multi-path refinement strategy can solve more tasks in the same CPU time than the single-path refinement strategy does. The quantile plot in Figure 4.3 gives on its abscissa the number of solved tasks, that is, the number of verification tasks that cause at least one refinement and result in a *true* verdict. A data point indicates which quantile (value on abscissa) of the runs needs less than the given measure (in this case time in seconds on the ordinate axis) [BLW17].

From the plot we can see that at any time the multi-path refinement strategy (ranker: $\mathcal{R}_{ID}$, selector: $\mathcal{S}_\cup$, plotted in blue colour) has solved more verification tasks than the single-path strategy (ranker: $\mathcal{R}_F$, selector: $\mathcal{S}_F$, plotted in green colour). Within the time limit of 300 s, the single-path refinement strategy can successfully solve 130 tasks whereas the multi-path strategy is able to solve 150 verification tasks. Both curves end at around 250 s; the remaining time is needed for analysis set-up, parsing of the input files etc.

**Summary (*Hypothesis 2*):** We see that the multi-path refinement strategy cannot only solve more verification tasks overall but also solves more tasks at any time if we would set the run-time limit to such value. In conclusion, we **accept Hypothesis 2**.

**Table 4.6.** Number of relevant tasks per property for Hypothesis 3

| 08_1a | 10_1a | 32_1a | 43_1a | 68_1a | 68_1b | 77_1a | 101_1a | 106_1a | 118_1a | 129_1a | 132_1a | 134_1a | 147_1a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 171 | 108 | 163 | 120 | 169 | 181 | 86 | 186 | 163 | 0 | 6 | 152 | 86 | 187 |

### 4.6.2. *Influences of the Verification Tasks*

In the previous section we provided a comparison based on verification tasks with all 14 properties combined. We are now interested in the influences of the verification tasks. Therefore, we again use the multi-path and the single-path refinement strategy like before. We furthermore use the same selection of the 250 Linux kernel modules. We do this to get comparable results; not only for results relevant in this section but also to compare the results with those from the previous section. Thus, the only variable we change in the experiments here is the used property. For each verification task we combine the kernel module with just one property; thus the number of verification tasks is now 250 kernel modules combined with 14 different properties, that is, 3 500 verification tasks.

Not all properties are relevant for each tasks, obviously. Table 4.6 provides an overview for how many of the 250 kernel modules a specific property is relevant, the verification framework yields a *true* verdict for them, and at least one refinement was necessary.

Hypothesis 3 indicates the expectation that independent of the used property in a verification task the multi-path refinement strategy is able to outperform the single-path strategy in terms of use analysis CPU time. From the box plots in Figure 4.4 we can already see that this is not the case. Obviously, there are properties for that the hypothesis holds, most prominent the properties LDV_106_1a and LDV_132_1a. For most properties there is no significant speedup or slow-down for large parts of the verification tasks; only outliers exist. Noticeable is the slow-down for the properties LDV_08_1a, LDV_32_1a and LDV_43_1a. Verification tasks consisting of a kernel module and one of these properties can overall be solved faster by the single-path refinement strategy. Note that due to a bug in the property LDV_118_1a there were only *false* verdicts, hence, we omit this property for the further presentation. Note further, that property LDV_129_1a is only relevant with a *true* verdict for six kernel modules, which is too less to give any qualified insights for this property.

Due to the fact that there is no significant change in terms of analysis CPU time for many of the properties, we restrict our presentation to those with considerable changes. We are particularly interested in reasons for the shifts in consumed analysis CPU time.

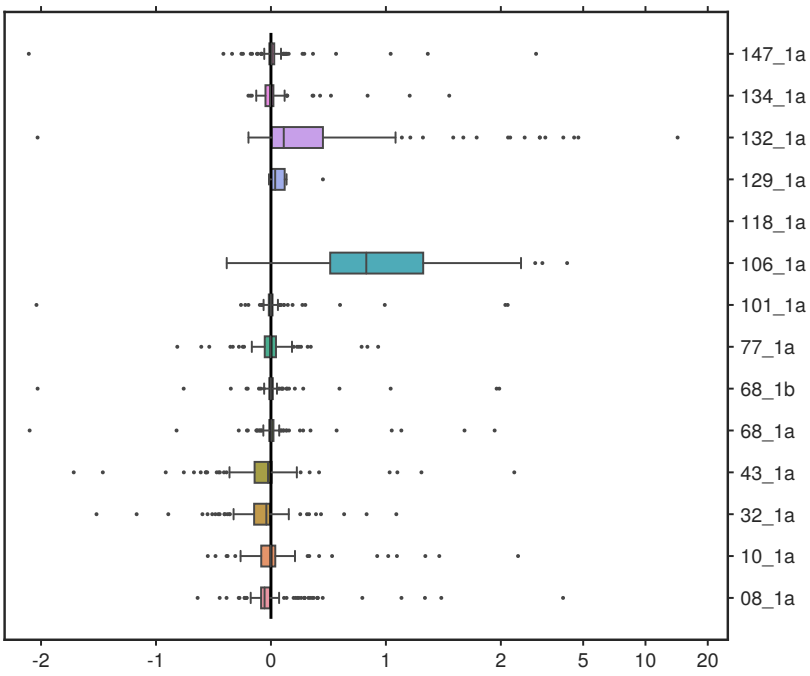


**Figure 4.4.** Comparison of speedups per property. Data points left of the vertical bar indicate that the single-path strategy used less CPU time than the multi-path strategy for the analysis; the opposite holds for data points right of the vertical bar. Note that a speedup of 1 in the figure is a speedup of 100 %, that is, execution time of the faster configuration is only half of the time the slower configuration takes.

Slowdowns As an example for the slowdown we take the results achieved with property LDV_32_1a. We provide a scatter plot showing the performance shifts in terms of analysis CPU times in Figure 4.5. Again, on the abscissa we see the single-path refinement strategy's values and on the ordinate axis the values of the multi-path refinement strategy.

We can clearly see that only few data points are in the lower right triangle, that is, data points for which the multi-path strategy consumes less analysis CPU time. A large portion of the data points is clustered near the origin of the axes. They represent verification tasks where both the single-path and the multi-path strategy can solve the task by only one refinement step. Hence, we cannot expect large speedups in either direction. The remaining data points indicate a faster single-path strategy.

Speedups For other properties we can determine speedups from Figure 4.4, most notably for the properties LDV_132_1a and LDV_106_1a. Detailed scatter plots showing the performance shifts in terms of analysis CPU times are provided; for the property LDV_106_1a in Figure 4.6 and for property LDV_132_1a in Figure 4.7.

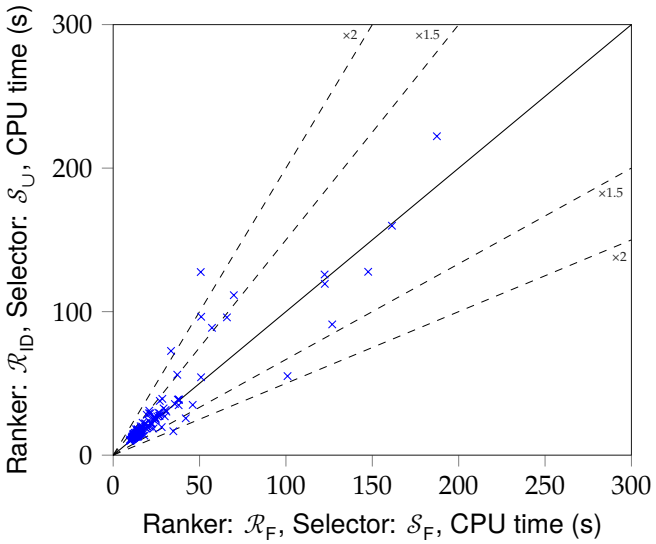


**Figure 4.5.** Analysis CPU time comparison for property LDV_32_1a

For property LDV_106_1a we encounter a slowdown for only one out of 163 tasks (that is 0.613 %). All other verification tasks result in a speedup with an average value of 1.99 and a median of 1.83. The best achieved speedup is 5.18.

The reason for these speedups lies again in the number of necessary refinements. The single-path strategy in sum takes 1 211 refinements, while 180 refinements are sufficient for all tasks with the multi-path strategy. The median number of refinements necessary for a verification task is 7 for the single-path strategy and 1 for the multi-path strategy. At most, the single-path strategy needs 40 refinement steps for one verification task, while the maximum number of refinement steps for one verification task taken by the multi-path strategy is only 3.

For property LDV_132_1a we encounter a slowdown for 31 of 152 tasks (that is 20.4 %). The other verification tasks result in a speedup with an average value of 1.52 and a median of 1.11. The best achieved speedup is 15.3.

Again, the speedups are caused by the smaller number of refinements. The single-path strategy takes a total of 567 refinements, while 171 refinements are suffcient for all tasks with the multi-path strategy. While the median number (2 for single-path, 1 for multi-path) does not differ too much, the single-path strategy causes at most 45 refinements for one verification task, while at most 3 refinements for one verification task are sufficient for the multi-path strategy.

In both cases, due to the smaller number of necessary refinements, time can be saved during precision adjustment as well as during the re-exploration of the state space after a refinement. Notwithstanding, such effects can only be used if a larger number of refinements is necessary, as we have seen, for example, for the property LDV_32_1a. It can be

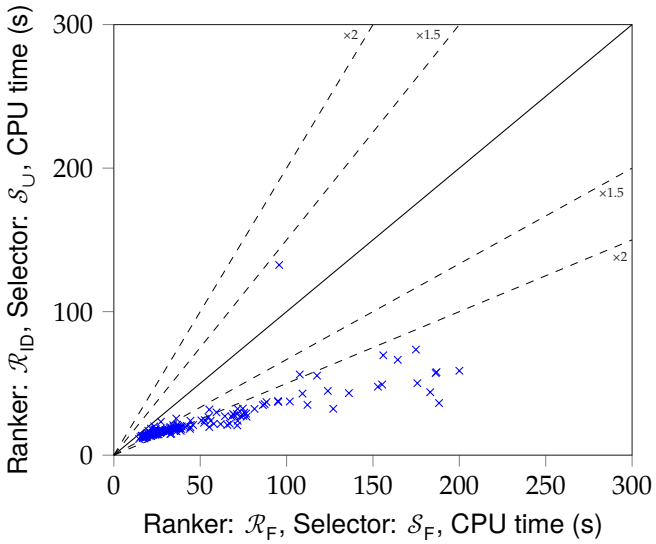


**Figure 4.6.** Analysis CPU time comparison for property LDV_106_1a

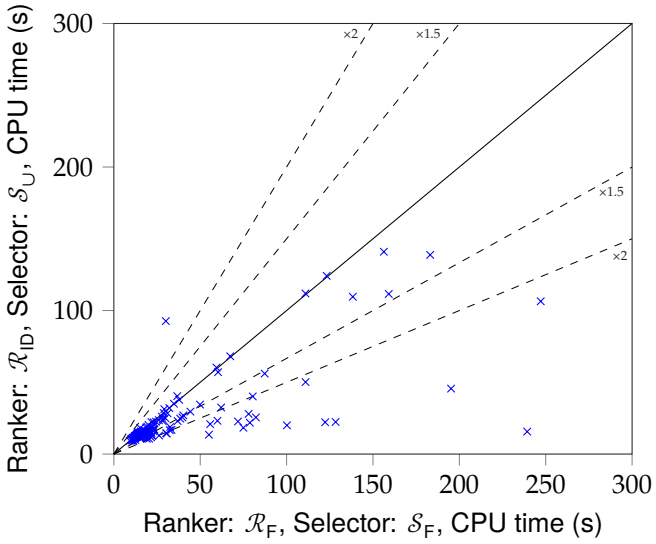


**Figure 4.7.** Analysis CPU time comparison for property LDV_132_1a

seen that the influence of the multi-path strategy largely depends on the used property.

**Summary (*Hypothesis 3*):** Whether a speedup can be achieved depends strongly on the property. A property that causes only one refinement for most of the tasks does not allow the multi-path refinement strategy to benefit from its information plus. For properties that cause larger numbers of refinements with the single-path strategy, the multi-path strategy can profit from its information plus. Although, for some properties the multi-path strategy is faster, it cannot outperform the single-path strategy in all cases. In conclusion, we **reject Hypothesis 3**.

### *4.6.3. Incorporating Domain Knowledge*

We have discussed the efficiency and effectiveness increases as well as the influence of the verification tasks on the analyses performance in the previous sections. In this section, we will focus on the influence of domain knowledge.

To get further insights, we use a multi-property approach again with all 14 properties combined with the 128 Linux kernel modules for that at least one refinement is necessary and the verification verdict is *true*. We formulated the expectation in Hypothesis 4 that by preferring target states with a smaller state level we can save refinements compared to preferring target states with a larger state level. Reason is that we expect the removal of more paths to target states by preferring short paths. In our presentation we also give the number of refinements caused by the single-path refinement strategy as well as by the multi-path refinement strategy for comparison reasons.

Because we know that we can safe most refinement steps by always combining all available information into a refinement, which is done by the multi-path strategy, we expect this strategy to perform best. All considered operator configurations where we sort the target states using an appropriate ranking operator do combine the precision of two to eight provident refinements. We present the number of refinements for each operator configuration in Figure 4.8.

Looking at the numbers, note that incidentally the median number of refinements is always the maximum of the third quartile, that is, the right border of the box. Furthermore, we notice that the overall number of refinements is quite small and most of the verification tasks can be solved with at most five refinements even for the single-path refinement strategy. Nonetheless, it is possible to save refinements when preferring target states with a smaller state level.

**Summary (*Hypothesis 4*):** Preferring target states with a smaller state level over such with a larger state level saves at least a few refinements. However, the overall number of refinements is not very large for most of the verification tasks, thus we cannot expect large savings in terms of refinement steps. Nevertheless, in conclusion, we **accept Hypothesis 4**.

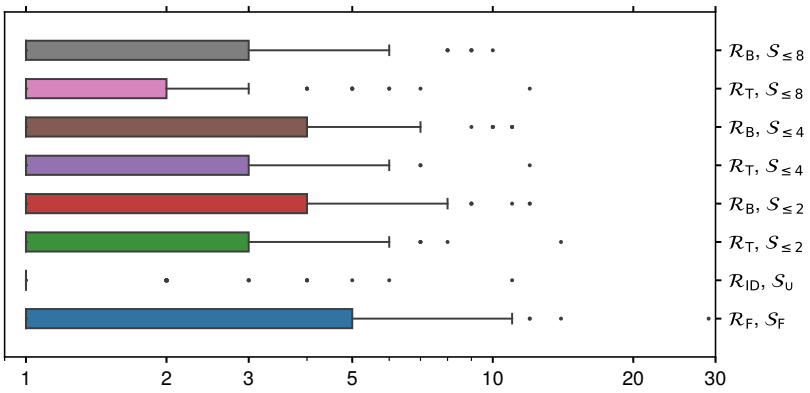


**Figure 4.8.** Number of applied refinements for the different operator configurations. Haphazardly, the median value of all configurations is always the maximum of the third quartile, that is, the right border of the box.

## 4.7. Discussion

Our work studied different perspectives of the performance behaviour of a multi-path refinement strategy. We were especially interested whether we can benefit from our multi-path strategy when verifying a real-work software system. For that we used 250 Linux kernel modules and 14 safety properties that were also used in previous work [ABM+16].

The results show a differentiated picture. We accepted both Hypotheses 1 and 2, which gave us an insight on the effectiveness and efficiency of our multi-path refinement strategy compared to a single-path strategy. Multi-path refinement increases efficiency in terms of analysis CPU time with an average speedup of 1.71 and a median speedup of 1.47. In terms of effectiveness we can say that our multi-path strategy strictly performs better than the single-path strategy. Thus, we can given an answer to Research Question 1: there is a multi-path refinement strategy that performs better than a single-path strategy. The multi-path strategy consists of an identity ranker and an all-join of the precisions that are found during the provident refinement steps.

We rejected Hypothesis 3 because our multi-path strategy did not outperform the single-path strategy for all properties. This leads to the result that the characteristics of the verification task has a considerable impact on the performance of the analysis, which answers Research Question 2.

Furthermore, we investigated in the influence of domain knowledge. We accept Hypothesis 4, although we notice that our case study does not provide tasks that cause a large number of refinements. It leads to the result that we can answer Research Question 3: incorporating domain knowledge in the analysis can pay off and save refinements.

Our results give a promising insight that extending refinement procedures to use the information of more than one counterexample path

during a refinement can lead to a improvement. We evaluated our approach using Linux kernel modules, a real-world software system, which makes our claims independent from artificial benchmark sets that where only constructed to show the alleged effects.

## 4.8. Threats to Validity

We are aware that no matter how careful one designs an empirical study there always are circumstances that can invalidate the study's results. We face these threats and name countermeasures in this section.

### *4.8.1. Internal Validity*

Since our analysis is based on predicate abstraction and uses a SMT solver for handling the predicates the choice of this solver influences the results. For our experiments we only use the MathSAT 5 SMT solver to make our results comparable. Nevertheless, we cannot obviate that the results differ significantly when using another SMT solver. Likewise does the JVM and its JIT influence the results. We try to minimise those effects by our benchmark setup; for example, we let the JVM pre-compile all code to minimise such influences.

Moreover, we are not able to change characteristics of our verification tasks. Control-flow and data dependences or the structure of the kernel modules and the properties cannot be changed for the selected case study, even if they cause variables that should be controlled or considered independently.

### *4.8.2. External Validity*

Generalising the results of our study is difficult due to the choice of our subject systems. We only use a small set of Linux kernel modules, which were selected randomly from the available modules of Linux 4.0-rc1. Hence, we can only make claims for this set of kernel modules. It is likely, that similar results can be achieved also for a wider selection of Linux kernel modules because they have similar structures. Nonetheless, it is impossible to claim this without further evaluation, which is not part of the scope of this work.

The results, however, are definitely not generalisable to other tasks.

Furthermore, the properties used by us are specific to the Linux kernel. They model expected behaviour of the kernel modules, which is very specific and would not be expected in other software systems.

# 5

# Related Work

We give an overview over work that is related to ours in this chapter.

## 5.1. Related Approaches

Closest to our work is the work of Beyer, Löwe, and Wendler on refinement selection [BLW15b]. For a path they do more than one interpolation query and then select their refinement based on heuristics. Such heuristics are, for example, that they prefer normal program variables over loop counters. In contrast to our work they only focus on heuristics to select refinements for only *one* error path and base their implementation on a standard CEGAR algorithm. Furthermore, we do not rely on the principle of lazy abstraction [HJM+02] but do a restart after each refinement, which is not done for refinement selection.

Lazy abstraction [HJM+02] is a key concept for many software model-checking tools. The technique is implemented in tools like, for example, Blast [HJM+02; HJM+03], which is also based on CEGAR and predicate abstraction. The idea is to stop the state-space exploration once an error location is found and immediately do a refinement; the exploration is continued after the successful refinement. This leads to early removal of spurious counterexamples and has been successfully used in practice. In contrast, our framework does not stop state-space exploration as soon as an error location is reached but always explores the full reachable state space.

Impact [McM06] is an algorithm that is also CEGAR-based. It creates an unwinding of the CFA where abstract states are labeled with formulæ over the program variables. It never computes abstractions and initializes all new abstract states to *true*. This is similar to our framework—and to predicate-abstraction algorithms in general—while the precision is still empty [Wen17]. The Impact algorithm does also use interpolants for refining but it is based on lazy abstraction, which differs from our framework.

IC3 [Bra11] is an algorithm for model-checking finite-state systems. It is also known as *property-directed reachability* (PDR) [EMB11]. The algorithm produces inductive invariants until they are strong enough to prove safety of the program. It achieves this by incremental clause learning, where new clauses are inductive with respect to the previously

learned clauses. These clauses are derived from counterexamples. PDR can be understood as a different strategy for discovering predicates during refinement, whereas we use Craig interpolation. The IC3 algorithm can be extended to infinite-state systems by CTIGAR [BBW14], which is an SMT-based IC3 algorithm incorporating CEGAR. A tool implementing this CTIGAR approach for software model checking is, for example, the Vienna Verification Tool [GLW16].

An automaton-based approach on model checking is presented in the work of Heizmann, Hoenicke, and Podelski [HHP09]. A program is considered a set of traces, which are words over the alphabet of the CFA. Starting with the CFA as initial abstraction they construct further abstractions in a CEGAR-style refinement loop [HCD+17]. The infeasibility of a trace is iteratively proven by interpolation and represented by Hoare triplets [Hoa69]. From an infeasible trace a interpolant automaton is created that does not only accept the given error trace but many other infeasible traces, too. The program is correct, if the Hoare triplets are valid for every control flow trace [HHP13]. This approach is implemented in the tool Ultimate Automizer [HCD+13; HDG+16; HCD+17]. Common with our technique is that the approach does not perform lazy abstraction but explores the full state space. In contrast to them, our approach is not centred on abstract traces but on abstract states. Nevertheless, similarities are in the fact that we treat the precision gained from the interpolants as independent from states, which is achieved by the interpolant automata in their work.

## 5.2. Types of Model Checking

Model Checking dates back to the 1970's beginning with the concept of concrete enumerative model checking. The core idea is to traverse a graph of program states and transitions where each program state is represented exactly. Manipulations are done on individual states, as opposed to symbolic model checking, which manipulates sets of states. The technique grew out of testing and simulation [Sun78; STE+82].

Execution-based model checking is a special case of the enumerative principle. It uses the runtime systems of programming language runtimes for an enumerative state space exploration. Tools like Verisoft [God97] or JavaPathFinder [VHB+03] implement this approach.

In contrast to the enumerative concepts, symbolic model checking represents the program as sets of states. *Bounded Model Checking (BMC)* unrolls the control-flow graph for a fixed number of steps and checks reachability of error locations within this number of steps [BCC+99]. It is also related to symbolic execution [Kin76]. Well-known bounded model checking tools are, for example, CBMC [CKY03] or Calysto [BH08].

Several approaches were proposed for unbounding. One such approach is called *k*-induction [SSS00; MRS03], which uses inductive invariants. It is, for example, implemented in CPAchecker [BDW15].

Furthermore, invariant generation, incremental bounded model checking, and incremental *k*-induction have been combined, for example, in the tool 2LS [SK16].

Further techniques use abstract models for infinite state programs. They use a reachability analysis on different abstract domains to capture only necessary information [CC77]. In order to iteratively increment the captured information, algorithms like CEGAR [CGJ+00; CGJ+03] have been proposed. Such approaches are implemented, for example, in tools like SLAM [BR02], BLAST [HJM+02; HJM+03], or CPACHECKER [BK11]. The latter two extend the CEGAR approach further by using lazy abstraction [HJM+02].

## 5.3. BENCHMARKS

Unfortunately, many authors do not provide empirical studies or at least some benchmark results on the success of their approaches.

An established benchmark set is provided by the International Competition on Software Verification (SV-COMP) [Bey17]. It provides more than 8 900 verification tasks consisting of a C program and a property modelling reachability, memory safety, or termination. Regrettably, many of those tasks are artificially created and not derived from real-world software systems. There is, however, a category of Linux kernel modules and programs from the BUSYBOX suite[1]. The competition does not perform extensive empirical studies but introduces a custom score schema, where a value gets assigned to correct and incorrect verifier verdicts, which will then be accumulated to a resulting score—achieving the highest score is considered to be best.

An extensive evaluation using a large corpus of verification tasks derived from a real-world software system is used by Apel et. al. [ABM+16]. They use 4 336 Linux kernel modules, from which we use the presented subset of 250 modules, and provide 14 safety properties, which we also use.

[1] BUSYBOX is a collection of many common UNIX utilities into a small executable. They are especially optimised for small and embedded systems. See busybox.net for more details.

# 6

# Conclusion

Let us conclude this work with a summary and a prospect on possible future work and research ideas.

## 6.1. Summary

We presented a novel approach and framework for provident abstraction refinements. Our framework extends traditional CEGAR-based approaches and allows us to incorporate information about the precision gained from the provident refinement step in the selection of an executed refinement. With this information we are able to delay, estimate, combine, and rank the possible refinements.

After introducing the approach and formally define it, we conducted an empirical study to get insights whether the approach is promising. Thereto, we implemented our approach in a fork of the open source model-checking framework CPAchecker. Our study is based on 250 modules from the Linux kernel together with 14 safety properties that model correct behaviour of the kernel modules.

The results are promising and we were able to show that the multi-path refinement approach is beneficial for efficiency as well as effectiveness in terms of analysis CPU time. Especially in cases where a single-path refinement strategy causes large numbers of refinements, the multi-path refinement strategy can be beneficial. This is due to the fact that we incorporate the information of many possible refinements and combine it, such that there is no need to execute them one after another. This is possible because the calculation of the new precision information is extremely cheap compared to exploring the abstract state space or adjusting the precision of the states.

We concluded our presentation with a discussion of threats that challenge the validity of our study and gave an overview over related work.

## 6.2. Future Work

Solving a research problem almost always leads to new problems and further open questions. In this final section of the thesis, we want to discuss a few such questions that arose during the work on the thesis.

We leave them as future work because they would blast the scope of this work.

We already stated in the discussion of Research Question 3 that our case study did not provide tasks causing large numbers of refinements. Likewise is the number of 250 kernel modules, which were selected randomly from a larger pool of kernel modules, quite small. Hence, extending the evaluation to a larger set of verification tasks is a natural first idea. Although the previous work from which we took the kernel modules [ABM+16] provides such a larger set of verification tasks, namely 4 336 Linux kernel modules, we decided to omit them due to time reasons. Furthermore, incorporating other case studies, such as programs from the BusyBox tool suite is an idea to gain further insights on the behaviour of our framework and make more general claims.

Model checking frameworks can also be used for test-case generation [HMR04; VPK04]. Using a CEGAR-based model checking framework together with a multi-goal analysis improves the performance of such an approach [BLB+15]. Our approach fits perfect into this concept because of our focus on multiple paths through the ARG. Thus, it would be interesting to combine the concepts of CPA/tiger [BHT+13; BLB+15] with our refinement framework.

# A

# Bibliography


[ABM+16] S. Apel, D. Beyer, V. O. Mordan, V. S. Mutilin, and A. Stahlbauer. "On-the-fly Decomposition of Specifications in Software Model Checking". In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016, Seattle, WA, USA, November 13–18)*. Ed. by T. Zimmermann, J. Cleland-Huang, and Z. Su. ACM, 2016, pp. 349–361 (*cited on pp. 2, 3, 8, 25, 26, 28, 29, 39, 43, 46*).

[BH08] D. Babic and A. J. Hu. "Calysto: Scalable and Precise Extended Static Checking". In: *Proceedings of the 30th International Conference on Software Engieneering (ICSE 2008, Leipzig, Germany, May 10–18)*. Ed. by W. Schäfer, M. B. Dwyer, and V. Gruhn. ACM, 2008, pp. 211–220 (*cited on p. 42*).

[BR02] T. Ball and S. K. Rajamani. "The SLAM Project: Debugging System Software via Static Analysis". In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002, Portland, OR, USA, January 16–18)*. Ed. by J. Launchbury and J. C. Mitchell. ACM, 2002, pp. 1–3 (*cited on p. 43*).

[Bey17] D. Beyer. "Software Verification with Validation of Results. Report on SV-COMP 2017". In: *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017, Uppsala, Sweden, April 22–29)*. Ed. by A. Legay and T. Margaria. Vol. 10206. Lecture Notes in Computer Science. Springer, 2017, pp. 331–349 (*cited on p. 43*).

[BDW15] D. Beyer, M. Dangl, and P. Wendler. "Boosting *k*-Induction with Continuously-Refined Invariants". In: *Proceedings of the 27th International Conference on Computer Aided Verification (CAV 2015, San Francisco, CA, USA, July 18–24)*. Ed. by D. Kroening and C. S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 622–640 (*cited on p. 42*).

[BDW18] D. Beyer, M. Dangl, and P. Wendler. "A Unifying View on SMT-Based Software Verification". In: *Journal of Automated Reasoning* 60.3 (2018), pp. 299–335 (*cited on pp. 6, 8, 9, 11, 12*).

[BHT07] D. Beyer, T. A. Henzinger, and G. Théoduloz. "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis". In: *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007, Berlin, Germany, July 3–7)*. Ed. by W. Damm and H. Hermanns. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 504–518 (*cited on pp. 3, 8, 9*).

[BHT08] D. Beyer, T. A. Henzinger, and G. Théoduloz. "Program Analysis with Dynamic Precision Adjustment". In: *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008, L'Aquila, Italy, September 15–19)*. IEEE Computer Society, 2008, pp. 29–38 (*cited on pp. 3, 8, 9*).

[BHT+13] D. Beyer, A. Holzer, M. Tautschnig, and H. Veith. "Information Reuse for Multi-Goal Reachability Analyses". In: *Proceedings of the 22nd European Symposium on Programming Languages and Systems (ESOP 2013, Rome, Italy, March 16–24)*. Ed. by M. Felleisen and P. Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 472–491 (*cited on p. 46*).

[BK11] D. Beyer and M. E. Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011, Snowbird, UT, USA, July 14–20)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 184–190 (*cited on pp. 7, 8, 43*).

[BL13] D. Beyer and S. Löwe. "Explicit-State Software Model Checking Based on CEGAR and Interpolation". In: *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013, Rome, Italy, March 16–24)*. Ed. by V. Cortellessa and D. Varró. Vol. 7793. Lecture Notes in Computer Science. Springer, 2013, pp. 146–162 (*cited on pp. 12, 16*).

[BLN+13] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler. "Precision Reuse for Efficient Regression Verification". In: *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2013, Saint Petersburg, Russian Federation, August 18–26)*. Ed. by B.

Meyer, L. Baresi, and M. Mezini. ACM, 2013, pp. 389–399 (*cited on pp. 12, 16*).

[BLW15a] D. Beyer, S. Löwe, and P. Wendler. "Benchmarking and Resource Measurement". In: *Proceedings of the 22nd International Symposium on Model Checking Software (SPIN 2015, Stellenbosch, South Africa, August 24–26)*. Ed. by B. Fischer and J. Geldenhuys. Vol. 9232. Lecture Notes in Computer Science. Springer, 2015, pp. 160–178 (*cited on p. 29*).

[BLW15b] D. Beyer, S. Löwe, and P. Wendler. "Refinement Selection". In: *Proceedings of the 22nd International Symposium on Model Checking Software (SPIN 2015, Stellenbosch, South Africa, August 24–26)*. Ed. by B. Fischer and J. Geldenhuys. Vol. 9232. Lecture Notes in Computer Science. Springer, 2015, pp. 20–28 (*cited on pp. 2, 10, 11, 13, 18, 21, 41*).

[BLW15c] D. Beyer, S. Löwe, and P. Wendler. "Sliced Path Prefixes: An Effective Method the Enable Refinement Selection". In: *Proceedings of the 35th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2015, Grenoble, France, June 2–4)*. Ed. by S. Graf and M. Viswanathan. Vol. 9039. Lecture Notes in Computer Science. Springer, 2015, pp. 228–243 (*cited on pp. 2, 13, 21*).

[BLW17] D. Beyer, S. Löwe, and P. Wendler. "Reliable Benchmarking: Requirements and Solutions". In: *International Journal on Software Tools for Technology Transfer* (2017). Pre-published (*cited on pp. 29, 33*).

[BCC+99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. "Symbolic Model Checking Using SAT Procedures instead of BDDs". In: *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference (DAC '99, New Orleans, LA, USA, June 21–25)*. Ed. by M. J. Irwin. ACM Press, 1999, pp. 317–320 (*cited on p. 42*).

[BBW14] J. Birgmeier, A. R. Bradley, and G. Weissenbacher. "Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR)". In: *Proceedings of the 26th International Conference on Computer Aided Verification (CAV 2014, Vienna, Austria, July 18–22)*. Ed. by A. Biere and R. Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 831–848 (*cited on p. 42*).

[Bra11] A. R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Proceedings of the 12th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2011, Austin, TX, USA, January 23–25)*. Ed. by R. Jhala

and D. A. Schmidt. Vol. 6538. Lectures Notes in Computer Science. Springer, 2011, pp. 70–87 (*cited on p. 41*).

[BLB+15] J. Bürdek, M. Lochau, S. Bauregger, A. Holzer, A. von Rhein, S. Apel, and D. Beyer. "Facilitating Reuse in Multi-Goal Test-Suite Generation for Software Product Lines". In: *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering (FASE 2015, London, UK, April 11–18)*. Ed. by A. Egyed and I. Schaefer. Vol. 9033. Lecture Notes in Computer Science. Springer, 2015, pp. 84–99 (*cited on p. 46*).

[CGS+13] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. "The MathSAT 5 SMT Solver". In: *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013, Rome, Italy, March 16–24)*. Ed. by N. Piterman and S. A. Smolka. Vol. 7795. Lecture Notes in Computer Science. Springer, 2013, pp. 93–107 (*cited on p. 29*).

[CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach". In: *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '83, Austin, TX, USA, January 24–26)*. Ed. by J. R. Wright, L. Landweber, A. J. Demers, and T. Teitelbaum. ACM Press, 1983, pp. 117–236 (*cited on pp. 1, 2*).

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". In: *ACM Transactions on Programming Languages and Systems* 8.2 (1986), pp. 244–263 (*cited on pp. 1, 2*).

[CGJ+00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-Guided Abstraction Refinement". In: *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000, Chicago, IL, USA, July 15–19)*. Ed. by E. A. Emerson and A. P. Sistla. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 154–169 (*cited on pp. 10, 43*).

[CGJ+03] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking". In: *Journal of the ACM* 50.5 (2003), pp. 752–794 (*cited on pp. 2, 10, 43*).

[CKY03] E. M. Clarke, D. Kroening, and K. Yorav. "Behavioral Consistency of C and Verilog Programs under Bounded Model Checking". In: *Proceedings of the 40th Annual Design Auto-*

*mation Conference (DAC 2003, Anaheim, CA, USA, June 2–6)*. ACM, 2003, pp. 368–371 (*cited on p. 42*).

[CM90] R. H. Cobb and H. D. Mills. "Engineering Software under Statistical Quality Control". In: *IEEE Software* 7.6 (1990), pp. 44–54 (*cited on p. 1*).

[CC77] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77, Los Angeles, CA, USA, January 17–19)*. Ed. by R. M. Graham, M. A. Harrison, and R. Sethi. ACM, 1977, pp. 238–252 (*cited on p. 43*).

[Cra57] W. Craig. "Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem". In: *The Journal of Symbolic Logic* 22.3 (1957), pp. 250–268 (*cited on p. 12*).

[Dah17] P. Dahlberg. "Assessing the Scope of Safety Properties". MA thesis. University of Passau, Germany, 2017 (*cited on pp. 22, 25*).

[Dij74] E. W. Dijkstra. "On the role of scientific thought". EWD447, published in [Dij82]. Aug. 1974 (*cited on p. 1*).

[Dij76] E. W. Dijkstra. "A Discipline of Programming". Prentice-Hall, 1976. ISBN: 013215871X (*cited on p. 5*).

[Dij82] E. W. Dijkstra. "Selected Writings on Computing. A Personal Perspective". Texts and Monographs in Computer Science. Springer-Verlag, 1982. ISBN: 978-3-540-90652-0 (*cited on p. 51*).

[EMB11] N. Eén, A. Mishchenko, and R. K. Brayton. "Efficient Implementation of Property Directed Reachability". In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD 2011, Austin, TX, USA, October 30–November 02)*. Ed. by P. Bjesse and A. Slobodová. FMCAD Inc., 2011, pp. 125–134 (*cited on p. 41*).

[GHJ+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. "Design Patterns. Elements of Reusable Object-Oriented Software". Addison-Wesley, 1995. ISBN: 0-201-63361-2 (*cited on p. 9*).

[God97] P. Godefroid. "Model Checking for Programming Languages using Verisoft". In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97, Paris, France, January 15–17)*. Ed. by P. Lee, F. Henglein, and N. D. Jones. ACM Press, 1997, pp. 174–186 (*cited on p. 42*).

[GS97] S. Graf and H. Saïdi. "Construction of Abstract State Graphs with PVS". In: *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97, Haifa, Israel, June 22–25)*. Ed. by O. Grumberg. Vol. 1254. Lecture Notes in Computer Science. Springer, 1997 (*cited on p. 12*).

[GLW16] H. Günther, A. Laarman, and G. Weissenbacher. "Vienna Verification Tool: IC3 for Parallel Software. Competition Contribution". In: *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Software (TACAS 2016, Eindhoven, The Netherlands, April 2–8)*. Ed. by M. Chechik and J. Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 954–957 (*cited on p. 42*).

[HMR04] G. Hamon, L. M. de Moura, and J. M. Rushby. "Generating Efficient Test Sets with a Model Checker". In: *Proceedings of the 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004, Beijing, China, September 28–30)*. IEEE Computer Society, 2004, pp. 261–270 (*cited on p. 46*).

[HCD+17] M. Heizmann, Y. Chen, D. Dietsch, M. Greitschus, A. Nutz, B. Musa, C. Schätzle, C. Schilling, F. Schüssele, and A. Podelski. "Ultimate Automizer with an On-Demand Construction of Floyd-Hoare Automata. Competition Contribution". In: *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017, Uppsala, Sweden, April 22–29)*. Ed. by A. Legay and T. Margaria. Vol. 10206. Lecture Notes in Computer Science. Springer, 2017, pp. 394–398 (*cited on p. 42*).

[HCD+13] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski. "Ultimate Automizer with SMTInterpol. Competition Contribution". In: *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013, Rome, Italy, March 16–24)*. Ed. by N. Piterman and S. A. Smolka. Vol. 7795. Lecture Notes in Computer Science. Springer, 2013, pp. 641–643 (*cited on p. 42*).

[HDG+16] M. Heizmann, D. Dietsch, M. Greitschus, J. Leike, B. Musa, C. Schätzle, and A. Podelski. "Ultimate Automizer with Two-track Proofs. Competition Contribution". In: *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016, Eindhoven, The Netherlands, April 2–8)*. Ed. by M. Chechik and J. Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 950–953 (*cited on p. 42*).

[HHP09] M. Heizmann, J. Hoenicke, and A. Podelski. "Refinement of Trace Abstraction". In: *Proceedings of the 16th International Symposium on Static Analysis (SAS 2009, Los Angeles, CA, USA, August 9–11)*. Ed. by J. Palsberg and Z. Su. Vol. 5673. Lecture Notes in Computer Science. Springer, 2009, pp. 69–85 (*cited on p. 42*).

[HHP13] M. Heizmann, J. Hoenicke, and A. Podelski. "Software Model Checking for People Who Love Automata". In: *Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013, Saint Petersburg, Russia, July 13–19)*. Ed. by N. Sharygina and H. Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 36–52 (*cited on p. 42*).

[HJM+04] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. "Abstraction from Proofs". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004, Venice, Italy, January 14–16)*. Ed. by N. D. Jones and X. Leroy. ACM, 2004, pp. 232–244 (*cited on p. 16*).

[HJM+02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. "Lazy Abstraction". In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002, Portland, OR, USA, January 16–18)*. Ed. by J. Launchbury and J. C. Mitchell. ACM, 2002, pp. 58–70 (*cited on pp. 2, 13, 41, 43*).

[HJM+03] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. "Software Verification with Blast". In: *Proceedings of the 10th International Workshop on Model Checking Software (SPIN 2003, Portland, OR, USA, May 9–10)*. Ed. by T. Ball and S. K. Rajamani. Vol. 2648. Lecture Notes in Computer Science. Springer, 2003, pp. 235–239 (*cited on pp. 41, 43*).

[Hoa69] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580 (*cited on pp. 5, 42*).

[JM09] R. Jhala and R. Majumdar. "Software Model Checking". In: *ACM Computing Surveys* 41.4 (2009), 21:1–21:54 (*cited on p. 5*).

[KMP+09] A. V. Khoroshilov, V. S. Mutilin, A. Petrenko, and V. Zakharov. "Establishing Linux Driver Verification Process". In: *Proceedings of the 7th International Andrei Ershov Memorial Conference on Perspective of Systems Informatics (PSI 2009, Novosibirsk, Russia, June 15–19)*. Ed. by A. Pnueli, I. Virbitskaite, and A. Voronkov. Vol. 5947. Lecture Notes in

Computer Science. Springer, 2009, pp. 165–176 (*cited on pp. 3, 28*).

[Kin76] J. C. King. "Symbolic Execution and Program Testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394 (*cited on p. 42*).

[MG11] Z. Majo and T. R. Gross. "Memory Systems Performance in a NUMA Multicore Architecture". In: *Proceedings of the 4th Annual Haifa Experimental Systems Conference (SYSTOR 2011, Haifa, Israel, May 30–June 1)*. Ed. by P. Ta-Shma, J. Moreira, and L. Shrira. ACM, 2011, 12:1–12:10 (*cited on p. 28*).

[McC04] S. McConnell. "Code Complete. A Practical Handbook of Software Construction". 2nd ed. Microsoft Press, 2004. 960 pp. ISBN: 978-0-7356-1967-8 (*cited on p. 1*).

[McM06] K. L. McMillan. "Lazy Abstraction with Interpolants". In: *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006, Seattle, WA, USA, August 17–20)*. Ed. by T. Ball and R. B. Jones. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006, pp. 123–136 (*cited on p. 41*).

[MRS03] L. M. de Moura, H. Rueß, and M. Sorea. "Bounded Model Checking and Induction: From Refutation to Verification". In: *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003, Boulder, CO, USA, July 8–12)*. Ed. by W. A. H. Jr. and F. Somenzi. Vol. 2725. Lecture Notes in Computer Science. Springer, 2003, pp. 14–26 (*cited on p. 42*).

[SK16] P. Schrammel and D. Kroening. "2LS for Program Analysis. Competition Contribution". In: *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016, Eindhoven, The Netherlands, April 2–8)*. Ed. by M. Chechik and J. Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 905–907 (*cited on p. 43*).

[SSS00] M. Sheeran, S. Singh, and G. Stålmarck. "Checking Safety Properties Using Induction and a SAT-Solver". In: *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000, Austin, TX, USA, November 1–3)*. Ed. by W. A. Hunt and S. D. Johnson. Vol. 1954. Lecture Notes in Computer Science. Springer, 2000, pp. 108–125 (*cited on p. 42*).

[Sun78] C. A. Sunshine. "Survey on Protocol Definition and Verification Techniques". In: *Computer Networks* 2 (1978), pp. 346–350 (*cited on p. 42*).

[STE+82] C. A. Sunshine, D. H. Thompson, R. W. Erickson, S. L. Gerhart, and D. Schwabe. "Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models". In: *IEEE Transactions on Software Engineering* 8.5 (1982), pp. 460–489 (*cited on p. 42*).

[Tur37] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. ISSN: 1460-244X (*cited on p. 5*).

[VHB+03] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. "Model Checking Programs". In: *Automated Software Engineering* 10.2 (2003), pp. 203–232 (*cited on p. 42*).

[VPK04] W. Visser, C. S. Pasareanu, and S. Khurshid. "Test Input Generation with Java PathFinder". In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004, Boston, MA, USA, July 11–14)*. Ed. by G. S. Avrunin and G. Rothermel. ACM, 2004, pp. 97–107 (*cited on p. 46*).

[Wen17] P. Wendler. "Towards Practical Predicate Analysis". PhD thesis. University of Passau, Germany, 2017 (*cited on p. 41*).

**Colophon:**

This document was processed using LuaLaTeX. The program's version info is: This is LuaTeX, Version 1.0.4 (TeX Live 2017). Fonts are TeX Gyre Pagella, TeX Gyre Heros, `Fira Code`, and some mathematical glyphs from Latin Modern Math. The document's template can be found on GitHub[1]. All figures are either created using Ti*k*Z or the Python libraries `matplotlib`, `numpy`, `pandas`, and `seaborn` combined with `matplotlib2tikz`.

[1] See github.com/stephanlukasczyk/latex-thesis-template

**Eidesstattliche Erklärung:**

Hiermit versichere ich an Eides statt, daß ich diese Masterarbeit 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 Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, 2018–03–08 _____

Stephan Lukasczyk