UNIVERSITY OF PASSAU
Department of Informatics and Mathematics

MASTER THESIS

# Predicate Granularity in Predicate Abstraction

*Author:*
Sebastian Böhm

September 18, 2019

*Advisors:*
Prof. Dr. Sven Apel
Prof. Dr. Gordon Fraser
Andreas Stahlbauer

**Sebastian Böhm**:
*Predicate Granularity in Predicate Abstraction*
Master Thesis, University of Passau, 2019

# Abstract

Predicate abstraction is used to make software verification practicable by reducing the size of the state-space that has to be explored to solve a certain verification task. Abstraction predicates define what information of the program under verification is retained in the abstract model and thus specify the level of abstraction.

During previous work, we discovered that the *granularity* of the abstraction predicates, that is their size and structure, influences the performance of predicate abstraction. While fine-grained predicates work well for some verification tasks, the abstraction costs will explode for other programs—the same applies to coarse-grained predicates.

In this work, we investigate the influence of the granularity of abstraction predicates on the performance of predicate abstraction. Therefore, we develop several strategies on how knowledge about a program's structure can be used to determine a suitable predicate granularity for the verification task at hand. We demonstrate the applicability of this approach with several scenarios and investigate whether these results generalize to a wider variety of programs.

# Contents

1 Introduction 1

2 Background 3
2.1 Program Representation . . . . . . . . . . . . . . . . . . . . . . 3
2.2 Abstraction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2.3 Abstraction Refinement . . . . . . . . . . . . . . . . . . . . . . 5
2.3.1 CEGAR . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.3.2 Craig Interpolation . . . . . . . . . . . . . . . . . . . . . 6

3 Predicate Granularity and Abstraction 7
3.1 The Abstraction Problem . . . . . . . . . . . . . . . . . . . . . 7
3.2 Predicate Granularity . . . . . . . . . . . . . . . . . . . . . . . 8
3.2.1 Predicate Granularity in CEGAR Algorithms . . . . . . 9

4 Strategies for Predicate Splitting 11
4.1 Split to Atoms . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
4.2 Never Split . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
4.3 Slicing-Based Predicate Splitting . . . . . . . . . . . . . . . . . 12
4.3.1 Slicing Using the PDG . . . . . . . . . . . . . . . . . . . 12
4.3.2 Slicing Using the CDG . . . . . . . . . . . . . . . . . . . 13
4.4 SSA-Based Predicate Splitting . . . . . . . . . . . . . . . . . . . 13

5 Evaluation 14
5.1 Research Questions . . . . . . . . . . . . . . . . . . . . . . . . 14
5.1.1 Suitability of the Split Strategies . . . . . . . . . . . . . 14
5.1.2 Efficiency and Effectiveness . . . . . . . . . . . . . . . . 15
5.2 Experiment Setup . . . . . . . . . . . . . . . . . . . . . . . . . 15
5.3 Sensibility Study . . . . . . . . . . . . . . . . . . . . . . . . . 18
5.4 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
5.4.1 Case Study SCENARIOS . . . . . . . . . . . . . . . . . . 20
5.4.2 Case Study SVCOMP . . . . . . . . . . . . . . . . . . . . 24
5.5 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
5.6 Threats to Validity . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.6.1 Internal Validity . . . . . . . . . . . . . . . . . . . . . . 29
5.6.2 External Validity . . . . . . . . . . . . . . . . . . . . . . 29

6 Related Work 31

7 Summary 33

**Bibliography** **34**

# List of Figures


2.1 Example program for predicate abstraction . . . . . . . . . . . . 5

5.1 Control-flow graphs for the SCENARIOS case study . . . . . . . . 17
5.2 Quantile plots of the analysis CPU time for the results of the sensibility study. . . . . . . . . . . . . . . . . . . . . . . . . . . 19
5.3 Results for the case study SCENARIOS grouped by scenario . . 21
5.4 ARG for scenario REACTIVE with split strategy *ATOMIC* (left) and *NONE* (right) . . . . . . . . . . . . . . . . . . . . . . . . . 23
5.5 Scatter plots for the case study SVCOMP . . . . . . . . . . . . . 24
5.6 Quantile plot of the number of refinements for the case study SVCOMP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
5.7 Quantile plot of the analysis CPU time for the SVCOMP case study. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
5.8 Quantile plots for selected categories of the SVCOMP case study. 27

# List of Tables

5.1 Number of solved tasks by category and split strategy. . . . . 28

# Chapter 1

# Introduction

The heavy-work of modern software verification tools is usually done by satisfiability solvers. The verification tool encodes the verification task into a formula that is then given to the solver of choice to determine a solution. As a consequence, the performance of the verification tool is heavily depending on the performance of the solver and, without intimate knowledge about its inner workings, it might be hard to optimize the performance of a verification tool. So what can we—who we see the solver as a black box—do to improve the performance?

At the heart of a verification algorithm often stands an abstraction procedure that transforms the potentially infinite state-space of a program into a finite abstract model on which the verification tool can then perform its analysis. In this work, we take a look at *Boolean predicate abstraction*—an abstraction technique that heavily relies on solver queries (in particular, it uses a potentially expensive `AllSAT` query)—and how we can improve the problem encoding such that the costs for the abstraction procedure are reduced.

During previous work, we encountered a scenario where splitting abstraction predicates into smaller parts caused the costs for the abstraction computations to explode. The reason was that due to the smaller predicates, the formula used in the abstraction procedure had an exponential number of models that need to be enumerated by the `AllSAT` query. As a result, the costs for the abstraction procedure grow more than exponentially with the problem size. In a different case, however, such fine-grained predicates were beneficial for the performance of predicate abstraction and keeping the predicates intact lead to much longer running times. Obviously, none of both strategies to shape the abstraction predicates is suitable in every case.

These observations lead to the notion of *predicate granularity*. This term captures both, the *size* and the *structure* of abstraction predicates. In this work, we explore how the granularity of predicates influences the performance of predicate abstraction and thus, the verification procedure around it. To get a better understanding of the context of this problem, we characterize the problem to find an abstraction of a program summary such that the underlying verification task can be solved efficiently with the term *abstraction problem*. The abstraction procedure is usually not aware of the nature of the underlying task, so we need to provide it with the necessary information. This is usually done in the form of an abstraction precision which, in the case of predicate abstraction, happens to be a set of abstraction predicates. These

predicates are usually obtained in an automated way, for example, using Craig-interpolation. However, the granularity of the generated predicates can still be changed afterwards and additional information about the verification task, like the trace of an infeasible counterexample or simply control- or data-flow dependencies of the program under verification, can be used to guide this process.

**Contributions** In this work, we make the following contributions:

- We introduce the concept of *predicate granularity* and discuss its role in predicate abstraction.
- We introduce the notion of an *abstraction problem* that builds the framework for discussing the relationship between predicate granularity and predicate abstraction.
- We develop several strategies for determining the granularity of predicates based on additional information extracted from the program under verification and provide implementations of these strategies based on the verification tool CPAchecker.
- We perform an experimental study to show that our strategies outperform the current predicate split strategies of CPACHECKER in certain situations and investigate whether these results generalize to a wider variety of programs.

**Overview** The rest of this thesis is structured as follows: In Chapter 2, we give some background information about software model checking, predicate abstraction, and abstraction refinement. Chapter 3 introduces the abstraction problem and the concept of predicate granularity and discusses the relationship between these two. The following Chapter 4 presents several strategies for determining the granularity of predicates along with notes on their implementation and integration in CPACHECKER. The applicability of these strategies is investigated in an experimental study in Chapter 5. In the end, we discuss some related work in Chapter 6 and conclude with Chapter 7 containing summarizing thoughts.

# Chapter 2

# Background

In this chapter, we provide background information about the most important concepts referenced throughout this work. Where applicable, we use the notation used in [21].

## 2.1 Program Representation

We denote a program as a tuple $P = (X, L, l_0, T)$ consisting of a set of variables $X$, a set of program locations $L$ with a designated entry location $l_0 \in L$ and a transition relation $T$. The elements $\tau \in T$ are tuples $(l, \rho, l')$ with $l, l' \in L$ and $\rho$ is a constraint over the variables in $X \cup X'$ where $X$ and $X'$ are the values of the variables at location $l$ or $l'$ respectively. A program defined as above forms a deterministic, finite automaton with the program locations as its states and the program transitions as its transitions. The initial state of the automaton is the program's entry location $l_0$. The transitions of the automaton are labeled with the constraints $\rho$ as defined by the transition relation. Such an automaton is called the *control-flow automaton* (CFA) of a program.

In this work, we look at software model checking based on reachability analysis. The task of software model checking is to determine whether a program satisfies a given property. A property can be encoded by introducing a special error location $\mathcal{E}$ into the program's transition system that can be reached if and only if the program violates that property. The model checking algorithm must then check if this error location $\mathcal{E}$ is reachable in this transition system or not. This type of analysis is called *reachability analysis*.

## 2.2 Abstraction

For programs with an infinite state-space, model checking algorithms based on reachability analysis may not terminate. To circumvent this problem, the reachability analysis is performed using an abstract model of the real program. The abstract model overapproximates the real program, meaning that every path through the real program has a counterpart in the abstract model (the converse is not necessarily true). As a result, the analysis is still sound but it loses precision, i.e., the analysis may report error paths that are infeasible in reality.

This kind of reachability analysis is called *abstract reachability analysis* and it is a form of *abstract interpretation* [10]. In abstract interpretation, an abstract model of the real program is created using an *abstract domain* $D = (C, \mathcal{E}, [\![\cdot]\!])$, where $C$ is the set of concrete states, $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ is a join-semi-lattice over the set $E$ of abstract states and $[\![\cdot]\!] : E \mapsto 2^C$ is a concretization function mapping abstract states to sets of concrete states. The abstract domain satisfies the properties $[\![\bot]\!] = \emptyset$ and $[\![\top]\!] = C$. Also, for all lattice elements $e, e' \in E$ it holds that $[\![e \sqcup e']\!] \supseteq [\![e]\!] \cup [\![e']\!]$. The abstract reachability analysis then explores the state-space of the program under verification by summarizing parts of the program, abstracting these summaries and conducting the reachability analysis using these abstractions. Each of the summaries consists of a collection of paths through the program and thus, represents a set of concrete program states. These summaries are abstracted into *abstract states* that overapproximate the set of states represented by the summary itself. The exploration of the abstract state-space terminates whenever the abstract state for the current block $S_1$ is already entailed by another abstract state $S_2$, i.e., when $S_1 \sqsubseteq S_2$. In this way, an abstract reachability graph (ARG) is constructed.

**Predicate Abstraction** One commonly used abstract domain is the *predicate abstract domain* [1, 16]. Intuitively, the state space of a program is partitioned into a finite set of abstract states by a finite set $\Pi$ of predicates over the program's variables. An abstract model of a program with respect to the predicates in $\Pi$ can be constructed automatically in the following way. Given a formula $\psi$ summarizing some part of a program (e.g. a block formula) and a set of predicates $\Pi$, the predicate abstraction $\phi$ of $\psi$ with respect to $\Pi$ is the strongest formula over $\Pi$ such that $\psi \rightarrow \phi$, i.e. $\phi$ is an *overapproximation* of $\psi$. Such a formula $\phi$ can be computed by building the conjunction of all models of the formula $f = p \bigwedge_{\pi \in \Pi} \pi \Leftrightarrow v_\pi$ where each $v_\pi$ is a unique propositional variable [22]. This method requires an expensive `AllSAT` query to be solved. There are other predicate abstraction techniques that avoid the `AllSAT` query [17] or that trade computation cost for precision (e.g. cartesian predicate abstraction [3]), but these are not considered in this work.

**Example** Let's consider the program block depicted in figure 2.1. In this figure, the ovals are program locations and the arrows are transitions with their operations annotated. Operations in square brackets are assumptions that split the program's control-flow. Now assume we want to compute an abstraction at the orange marked location. The block formula $\psi$ for this program block looks like this:

$$\psi = (b_1 = 0 \wedge a_1 \neq 0 \wedge b_2 = 1) \vee (b_1 = 0 \wedge a_1 = 0 \wedge b_2 = b_1)$$

The subscripts on the variable names are the static-single-assignment (SSA) indices [12]. Let $\Pi = \{(\pi_1, a_1 \neq 0), (\pi_2, b_2 = 1)\}$ be our set of predicates. Note that we give each predicate a name for the sake of this example. Also, we have instantiated the predicates with the latest SSA indices. This results

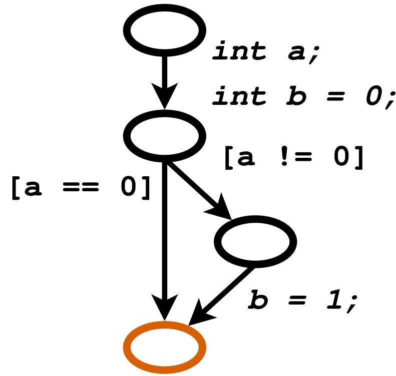



FIGURE 2.1: Example program for predicate abstraction

in the formula

$$f = \psi \wedge (\pi_1 \Leftrightarrow (a_1 \neq 0)) \wedge (\pi_2 \Leftrightarrow (b_2 = 1))$$

which has exactly two models, namely $\pi_1 \wedge \pi_2$ and $\neg\pi_1 \wedge \neg\pi_2$. Therefore, the predicate abstraction $\phi$ of $\psi$ with respect to $\Pi_1$ is:

$$\phi_1 = ((a_1 \neq 0) \wedge (b_2 = 1)) \vee (\neg(a_1 \neq 0) \wedge \neg(b_2 = 1))$$

## 2.3 Abstraction Refinement

As stated before, model checking an abstract model may report false errors caused by a too imprecise model. In that case, the abstract model needs to be refined. For a predicate abstraction based model checker, *counterexample-guided abstraction refinement* (CEGAR) in conjunction with a refinement procedure based on *Craig-interpolation* can be used to automate this process.

### 2.3.1 CEGAR

*Counterexample-guided abstraction refinement* (CEGAR) is a program verification algorithm capable of automatically refining an initially coarse abstract model until either the program is proven safe or a violation is found that is not an artifact of a too coarse model [9].

When used with predicate abstraction, CEGAR starts with constructing an abstract model using $\Pi = \emptyset$. In the first step, this model is given to the model checker which either can prove the property at hand or reports a violation along with a *counterexample trace*. This trace is a path through the abstract model that reaches an error location defined by the property. However, it might be that such a path does not exist in the underlying program. Therefore, in the second step of CEGAR, the counterexample is checked for feasibility. If it is found feasible, the violation is indeed valid and the algorithm terminates. Otherwise, the infeasibility proof is passed to the third step of CEGAR—the refinement step. Here, the counterexample is analyzed and new predicates are extracted and added to the set $\Pi$ such that at least

this counterexample is removed from the abstract model that can be constructed using the new $\Pi$. These three steps are repeated until either the abstract model—and therefore the corresponding program—is proven to adhere to the property at hand, or a true violation is found.

### 2.3.2 Craig Interpolation

We now take a more detailed look at the refinement step of the CEGAR algorithm. Henzinger et al. observed that the infeasibility-proof of a spurious counterexample trace encodes information about why it is infeasible [18]. They developed a method to extract additional facts (predicates) from the infeasibility proof that rules out at least that specific spurious counterexample. This method utilizes Craig's interpolation theorem [11] to extract the new predicates.

Given two formulas $\phi^-$ and $\phi^+$ whose conjunction is unsatisfiable, a *Craig-interpolant* for $(\phi^-, \phi^+)$ is a formula $\psi$ for which the following properties hold: (1) $\phi^- \Rightarrow \psi$, (2) $\psi \wedge \phi^+$ is unsatisfiable and (3) $\psi$ only uses variables common to $\phi^-$ and $\phi^+$. For a counterexample $l_0 \xrightarrow{\rho_0} l_1 \cdots l_{n-1} \xrightarrow{\rho_{n-1}} l_n$, the refinement procedure takes the counterexample's trace formula $\bigwedge_{i=0}^{n-1} \rho_i$ as an input. This formula is satisfiable if and only if the counterexample-path is feasible. For each state $l_i, i \in [1, n-1]$in the counterexample-path, the trace formula is split into two formulas $\phi_i^-$ and $\phi_i^+$ at the corresponding location and an interpolant for $(\phi_i^-, \phi_i^+)$ is constructed. These interpolants can then be transformed into abstraction predicates. The abstract model created with these additional new predicates is guaranteed to not contain the counterexample the predicates were extracted from.

# Chapter 3

# Predicate Granularity and Abstraction

In this chapter, we take a detailed look at the role of abstraction predicates and their granularity—that is, their size and structure—in predicate abstraction. We observed that the main performance bottleneck in predicate abstraction is the `AllSAT` query that is used to compute an overapproximation of the path formula and its more than exponential costs. As we will see, the granularity of predicates poses a degree of freedom in how we formulate the `AllSAT` query. We will discuss this concept of predicate granularity in order to come up with strategies for choosing a better granularity for the predicates. This chapter only gives an intuition about how such strategies could look like. For concrete implementations, we refer the reader to chapter 4.

## 3.1 The Abstraction Problem

Abstraction is a tool that is used because the problem at hand is too complex to be handled efficiently otherwise. The abstraction procedure is responsible for removing details from the problem input that are not needed for solving the underlying problem so that this problem becomes feasible. Because of that close connection between a problem at hand and an appropriate abstraction, we believe that it is important to look at abstraction not in isolation, but in the context of that accompanying problem. We call this configuration an *abstraction problem*.

An abstraction problem can be divided into three parts: an input summary, the underlying task and the abstraction procedure itself. The main issue is to determine which information of the input summary is needed to solve the underlying task and how can be ensured that this information is preserved in the abstraction. The abstraction procedure might not be aware of the underlying problem and therefore, does not know which information should be retained in the abstract model and which not. It is the responsibility of the user to guide the abstraction process and provide this information, for example in the form of an abstraction precision. Such information can, for example, be extracted from the input of the underlying problem. In the case of software verification, that input is computer programs that have a well-known structure and a lot of analysis and metrics are available to obtain all

kinds of information that might be helpful to the abstraction procedure and problem at hand.

In software model checking, abstraction is used to reduce the size of the state-space of the program under verification. The abstraction technique we take a look at is Boolean predicate abstraction. Our definition of an abstraction problem translates to this setting as follows: The underlying problem is a verification problem, that is, to decide whether a specific error state in the program's state-space is reachable or not. The abstraction procedure constructs abstract states by summarizing the concrete states inside a program block into an abstract state (cf. 2). The abstraction must ensure that an error location in the abstract model is reachable if and only if a corresponding error location is reachable in the real program. The abstract model's level of abstraction is defined by an abstraction precision, a set of predicates that is responsible for partitioning the concrete states. Typically, these predicates stem from a refinement procedure and include information necessary for correctly solving the verification task at hand.

## 3.2 Predicate Granularity

Let's look again at the example shown in figure 2.1 in the previous chapter. We used the abstraction precision $\Pi_1 = \{(\pi_1, a_1 \neq 0), (\pi_2, b_2 = 1)\}$ to compute an abstraction of the block formula $\psi$:

$$\psi = (b_1 = 0 \wedge a_1 \neq 0 \wedge b_2 = 1) \vee (b_1 = 0 \wedge a_1 = 0 \wedge b_2 = b_1)$$

The abstraction formula we constructed for this had two models. Now let $\Pi_2 = \{(\pi_1, a_1 \neq 0 \wedge b_2 = 1)\}$ be another set of predicates. The resulting abstraction formula $f_2$ has only one model, namely $\pi_1$, and the abstracted formula $\phi_2$ looks like the following:

$$\phi_2 = \pi_1 = (a_1 \neq 0 \wedge b_2 = 1)$$

One can easily see that different sets of predicates result in different abstractions. In particular, the abstraction constructed using $\Pi_2$ is more specific than the one constructed with $\Pi_1$. Also, note that the number of models of the abstraction formula is different for the abstraction precisions $\Pi_1$ and $\Pi_2$.

In the above example, the abstraction precision $\Pi_1$ can be constructed by splitting the predicate of $\Pi_2$ into a set of smaller—more fine-grained—predicates. During previous work, we observed that this difference in the *granularity* of the abstraction predicates influences how expensive the abstraction computation is. Depending on whether the abstraction precision consists of a few larger, coarse-grained predicates or more but finer-grained predicates, the performance of the abstraction procedure differed severely. In one case the analysis would complete in a matter of seconds, while in the other case the analysis would not terminate in a reasonable amount of time.

The example already hints at one important aspect: The predicate abstraction procedure relies on an `AllSAT`-query that has costs polynomial in

the number of models the formula in that query has. As this number grows exponentially with the number of predicates in the worst case, their granularity can have a severe influence on the performance of predicate abstraction. Intuitively, having very coarse-grained predicates should reduce the overall number of models as they tend to be larger, more complex, and fewer in number. However, very coarse-grained predicates may be too specific so that the verification task at hand cannot be solved with the resulting abstract model. In this case, other mechanisms are needed to counteract this problem (e.g. additional CEGAR iterations). Also, large and complex predicates add to the overall complexity of the abstraction formula what in turn increases the costs to solve the `AllSAT` query. More fine-grained predicates, on the other hand, give the abstraction procedure more flexibility in its task of over-approximating the given block formula. This is especially important in the presence of loops to find appropriate loop invariants. But, as demonstrated in the example above, very fine-grained predicates are prone to increasing the number of models of the abstraction formula leading to a more expensive `AllSAT` query.

There is a second aspect to our notion of predicate granularity besides predicate size, that is, the *structure* of the predicates. The structure of the abstraction predicates determines what information they encode that can help the analysis to converge faster towards an abstract model that does not contain any spurious counterexamples. The goal is to find a good compromise for the granularity of predicates that retains all relevant information for the verification task at hand while allowing the abstraction procedure to generalize. Strategies for achieving that goal are discussed in the next chapter.

### 3.2.1 Predicate Granularity in CEGAR Algorithms

Now that the notion of predicate granularity is introduced, this concept needs to be integrated into a suitable software verification algorithm. One such algorithm is CEGAR, a software verification algorithm that iteratively constructs, checks, and refines an initially coarse abstract model until no spurious counterexamples are encountered anymore. There are several different locations where the granularity of predicates can be decided in this algorithm. In the following, we discuss three possibilities along with their different advantages and shortcomings.

**During Abstraction** The obvious choice is to decide the predicate granularity right before the abstraction computation itself. This approach enables a per-abstraction decision about the granularity of the predicates, which offers the most flexibility. This is desirable if there is some information available about the concrete instance of the abstraction problem that is relevant to decide the best predicate granularity. But additional information about the verification task at hand, like for example the error location, might not be available at this point.

**During Refinement** In CEGAR, new predicates that can be used in future abstraction computations are generated during the refinement step. What makes this location attractive for determining the granularity of the predicates is that during refinement, we have access to the trace and the target state of a spurious counterexample—information that is important to solve the underlying verification problem. This information can be used to split the newly generated predicates into fragments of the desired granularity. However, this limits how predicates can be reused or shared with other program locations, as the granularity stays the same for all following abstraction computations.

**Hybrid Approaches** The previous approaches can also be combined to retain the flexibility of a per-abstraction decision about predicate granularity while still having access to the information available during refinement. Such an approach could store the additional information collected during refinement along with the coarse-grained predicates and then split the predicates on demand during the abstraction-phase. However, this needs additional implementation effort as well as memory to store the additional information.

In chapter 4 we present several strategies for predicate splitting that all work during the refinement step of CEGAR. We chose that location because our strategies rely on information about spurious counterexamples and their target states, information that is only available during refinement. Also, this helps to keep the implementation simple and clean.

# Chapter 4

# Strategies for Predicate Splitting

In the previous chapter, we identified the granularity of the abstraction predicates as a variable we can change and, in that way, influence the performance of predicate abstraction. In this chapter, we discuss several strategies on how predicates generated by the refinement procedure can be split into smaller fragments that are better suited for the abstraction problem at hand. We integrated these strategies into CPACHECKER and its predicate analysis as part of the refinement procedure. CPACHECKER is a software verification tool that is built on the concept of configurable program analysis and implements CEGAR, predicate abstraction and refinement based on Craig-interpolation [4, 5]. Our techniques are implemented in a fork of CPACHECKER that can be found on github[1]. The strategies are implemented as operators that take the produced interpolants as well as the current counterexample trace and target state as input and splits the interpolants into one or more predicates. It should be noted that no part of the interpolant is omitted to ensure the soundness of the operation.

## 4.1 Split to Atoms

One of the simplest possible strategies and currently the default configuration of CPACHECKER is to *always split* the predicates generated by the refinement procedure into their atoms making them as fine-grained as possible. This strategy shows good results in the presence of loops, as it inflicts the least restrictions when computing an abstraction, and therefore, exhibits the most potential for generalization (e.g. for finding loop invariants). However, as discussed before, because this strategy generates many small predicates, this also means that the formula constructed during abstraction tends to have more models resulting in a more expensive `AllSAT` query.

## 4.2 Never Split

Instead of splitting the predicates into atoms, they can also be left intact. This results in the coarsest granularity the predicates can have. Those predicates carry all the information needed to rule out the spurious counterexample the

[1] https://github.com/se2p/predicate-grinding

refinement was performed for. As these predicates are larger and more complex, it is less likely they can be combined to multiple models of a single block formula and so the problem of the previous split strategy can be avoided. However, the predicates generated with this strategy result in a lower level of abstraction so the explored state-space becomes larger. Additional refinement iterations might even be necessary to rule out spurious counterexamples that would not be contained in a model created more fine-grained predicates. Both cases lead to additional abstraction computations and an overall higher cost for solving the verification task. Also, as predicates become more complicated they increase the size and complexity of the abstraction formula which can also lead to a more expensive `AllSAT` query.

## 4.3 Slicing-Based Predicate Splitting

The predicates produced by the previous split strategies have the finest or coarsest granularity possible. We now explore several strategies that lie between those extremes and try to exploit additional information about the verification task at hand to determine the granularity of the abstraction predicates more intelligently.

The first two strategies rely on dependency information of the program under verification. The rationale behind these strategies is that reachability-based program verification is a path-sensitive analysis and therefore, information about the taken paths is most likely needed for a successful analysis. Thus, predicates describing program paths should be kept intact during splitting, while other information can be more fine-grained to allow better generalization.

### 4.3.1 Slicing Using the PDG

The first idea was to use program slicing to determine what dependencies should be preserved by the predicates after splitting. The program dependence graph (PDG)[15] of the program under verification is computed as a pre-processing step before the analysis starts. During refinement, we take the target state of the counterexample—the error location that was reached via an infeasible path—and compute a backward program slice [25]. This slice contains all the dependencies that can be encountered on the way towards this error location. Then all program variables contained in that slice are collected and the predicates are split such that sub-formulas consisting only of variables contained in the slice are left intact and all other parts of the predicates are split into their atoms. This is a heuristic that retains all data- and control-flow information in the predicates.

During the implementation, we encountered the problem that the program dependence graph generated by CPAchecker is not suitable for interprocedural slicing. Traditionally, interprocedural slicing is done using a system dependence graph [19], an extended form of the program dependence graph. As there is no system dependence graph available, we decided

to implement interprocedural slicing differently using call stack information. In CPACHECKER, this information is typically available along with the target location. The backward-traversal of the program dependence graph is then restricted to those paths that are compatible with the current call stack state truncating all function calls that could not have been taken in the current counterexample. This slicing technique, however, proved to be prone to producing too big slices. A slicing technique that fits this scenario better than traditional program slicing is path slicing [20], a technique specifically designed to extract all edges along some path through a program that are relevant to demonstrate the reachability of the target state of that program path. We achieve a similar behavior by restricting the slicing procedure to only follow edges where all the occurring variables are also contained in the interpolant. This simple optimization resulted in much smaller slices and greatly improved the results achieved with this split strategy.

### 4.3.2 Slicing Using the CDG

This predicate split strategy is essentially a further optimization of the previous strategy. For larger programs the data-flow analysis needed for computing the PDG can become very time- and memory-intensive. Also, for programs with many data-dependencies, the PDG can grow very large making program slicing rather expensive. In our experience, control-flow information is more relevant for the verification algorithm than data-flow information. With this in mind, we can simplify the PDG-based split strategy by performing the slicing only on the control dependence graph (CDG)—the PDG without data-dependencies. The implementation of this strategy is the same as for the PDG-based splitting strategy, only CPACHECKER's configuration is adjusted so that the PDG does not include data-dependencies.

## 4.4 SSA-Based Predicate Splitting

There is another way to extract dependency information from the counterexample-trace. To each state of the counterexample trace, there is a map with the static single assignment (SSA) indices of the program variables attached to it. By looking at the differences in the SSA-maps of two succeeding states along the counterexample trace we can determine which variables have changed. Here, we argue that for variables whose values change, we want to give the abstraction procedure more flexibility to generalize and thus, such predicates need to be more fine-grained. Other dependencies in the predicates model aspects of the program that do not change and therefore can remain as they are. Therefore, we split the parts of the formula that contain variables whose SSA-index has changed into atoms and leave the rest of the predicates intact.

# Chapter 5

# Evaluation

## 5.1 Research Questions

In the previous chapters, we discussed how the granularity of abstraction predicates can influence the performance of Boolean predicate abstraction and we designed and implemented several strategies for determining the predicate granularity in order to improve the performance of predicate abstraction. Now, we investigate the applicability of these predicate split strategies and whether our novel strategies lead to an increase in efficiency and effectiveness of the underlying software verification algorithm. We formulate these goals in the following research questions.

### 5.1.1 Suitability of the Split Strategies

We designed our predicate split strategies such that a specific abstraction problem can be solved more efficiently than when using the trivial split strategies, while still preserving all the information necessary for solving the underlying verification task. The first set of research questions investigates whether our novel strategies achieve these goals better than the existing trivial predicate split strategies.

#### RQ1.1: Which predicate split strategy produces the least number of models during predicate abstraction?

Predicate abstraction makes use of an `AllSAT` query to over-approximate block formulas. As `AllSAT` has costs exponential in the number of models of the formula, we expect a smaller number of models to result in improved performance of predicate abstraction. Our novel predicate split strategies aim to find a granularity for the predicates, such that the resulting formula used in the `AllSAT` query has fewer models than it would have with the default split strategies.

#### RQ1.2: Which predicate split strategy results in the least refinements?

Refinements tend to add additional costs to the analysis as for each refinement a new abstract model needs to be created, meaning that additional

abstraction computations are necessary. Therefore, a low number of refinements can be an indicator of a good predicate split strategy. Additional refinements may be necessary if the granularity of the predicates is too coarse so that the abstraction procedure is unable to deduce additional information not reflected in the predicates. In general, we expect to see fewer refinements the finer-grained the predicates are.

### 5.1.2 Efficiency and Effectiveness

The goal of our predicate split strategies is to increase the performance of predicate abstraction based program verification. Performance in this context means efficiency—how fast the algorithm can solve a verification task—and effectiveness—for how many verification tasks a result can be produced within certain resource limits.

#### RQ2.1: Which predicate split strategy is the most efficient in terms of CPU time?

The predicate split strategies presented in the previous chapter do come with additional costs. They all rely on dependency information of some sort, which is expensive to compute by itself—especially for large programs. Therefore, we need to investigate, whether the expected performance gains outweigh the costs for computing the predicate splittings.

#### RQ2.2: Which predicate split strategy is the most effective in terms of the number of solved tasks?

Because termination of the verification algorithm cannot be guaranteed, there exists a fixed time and memory budget for each verification task. When this budget is depleted, the task will be terminated with the result *UNKNOWN*. Therefore, an increase in efficiency usually goes along with an increase in effectiveness.

## 5.2 Experiment Setup

We perform our experiments using a custom version of CPACHECKER version 1.8 and its predicate analysis. The refinement procedure can be configured to use the different predicate split strategies as described in chapter 4. Also, we made some performance optimizations to the PDG generation algorithm. The code of this tool along with the used benchmark definitions can be found on github[1].

We divide this study into several experiments. Each experiment consists of a set of tasks. A task is defined as a tuple $T = (prog, sp, solver, blk)$, where

[1] https://github.com/se2p/predicate-grinding

*prog* is the program under verification, *sp* is the operator responsible for determining the granularity of the predicates, *solver* is the used SMT-solver and *blk* is the block operator.

The experiments are performed on machines with two Intel Xeon E5 processors running at 2.10*GHz* and 256*GB* of RAM running Debian 10 as an operating system. The tasks are executed using benchexec[2] in the version shipped with CPACHECKER. Each task is allowed to use two CPUs (according to the definition of benchexec, this means two virtual cores) and is restricted to 1800 seconds of CPU time and 25 gigabytes of RAM.

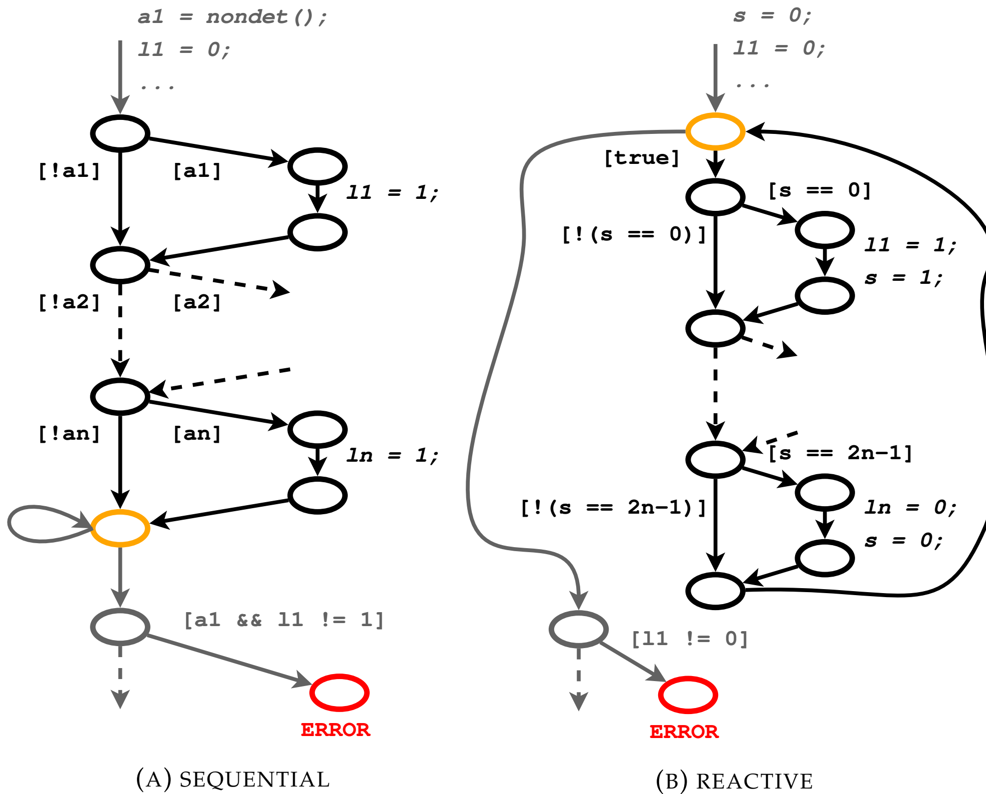
## Case studies

We perform our experiments with two case studies: a set of hand-crafted SCENARIOS that demonstrates the need and applicability of novel predicate split strategies and the larger SVCOMP program set which we use to investigate how the different strategies perform on a wider variety of programs.

**SCENARIOS** The SCENARIOS case study demonstrates worst- and best-case behavior of the trivial split strategies (*ATOMIC* and *NONE*) and shows how our novel split strategies can achieve best-case (or at least better) performance in all of the scenarios. Figure 5.1 shows the control-flow graphs for these scenarios. In this figure, the oval nodes are program locations and the arrows depict program transitions annotated with the corresponding program operations. Annotations in ***italic*** are assignments and annotations in **[square brackets]** are assumptions that split the control-flow into multiple paths. The orange node is the location where abstraction computations are performed, red nodes mark error locations. The important part of the control-flow graph is printed in black, parts of the graph shown in gray represent initialization code or encode the specification and are of secondary concern. All of the scenarios have a size parameter $n$ that can be scaled to simulate increasing complexity. In the control-flow graphs, the extension points are marked with dots or dashed lines. Experiments are performed with problem sizes ranging from 1 to 64.

The first scenario is called SEQUENTIAL and is shown in figure 5.1a. It is based on a benchmark used in [24] that models feature interactions in configurable systems. The program consists of $n$ independent choices creating $2^n$ different paths through the program. The specification checks whether the `if`-branches were executed correctly and thus makes the relationship between the `ai` and `li` relevant for the verification algorithm. Therefore, this information should be reflected in predicates with a good granularity for this scenario. If the predicates are split into atoms however, this results in $n$ predicates and $2^n$ possibilities to combine them into models the solver then has to enumerate during abstraction. Therefore, this scenario shall demonstrate why splitting predicates into atoms is not always a feasible strategy.

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




(A) SEQUENTIAL (B) REACTIVE

FIGURE 5.1: Control-flow graphs for the SCENARIOS case study

The second scenario REACTIVE models a finite state automaton with $n$ states. It consists of a loop inside which a variable representing the automaton's state is changed based on constraints modeling the automaton's transitions. In this case, each of the automatons states has exactly one successor, so the states form a loop themselves, that is run through over and over again. Inside this loop, a sequence of locks is acquired and then released again in the reverse order. Although technically never reachable, the specification checks that all locks are released upon the termination of the loop. The challenge in this scenario is to find an appropriate loop invariant that proves that the loop never terminates and thus, no error location is ever reachable. When not splitting the predicates at all, their number grows rapidly with the problem size adding to the complexity of the abstraction computation. This scenario shows that using the predicates as they are generated by the refinement procedure also will not work in every case.

The third and last scenario SEQ+RCT combines the other two scenarios by means of sequential composition. So first the SEQUENTIAL and immediately afterward, the REACTIVE verification problems have to be solved. As this scenario includes the challenges of both of the other scenarios, neither of the two default predicate split strategies ("split into atoms" and "do not split at all") will be able to solve larger instances of this scenario. Our novel predicate split strategies, however, should be able to handle this combination of opposing extremes at once.

**SVCOMP** This case study consists of a subset of the verification tasks used in the SV-COMP 2019 competition on software verification[3]. This collection of verification tasks was compiled by various contributors from the software verification community and contains programs from different domains with different characteristics. For this case study, we use the programs from the category *ReachSafety* except for the sub-category *Float* as not all of the used solvers support floats and the sub-categories *Heap* and *Recursive* as those require analysis techniques that are not or only partially supported by CPACHECKER in the desired configuration. It remains a set of 3080 programs distributed over seven sub-categories.

### Predicate Split Operators

In chapter 4, we described several strategies for determining the granularity of abstraction predicates. The strategy *ATOMIC*—CPACHECKER's default behavior—splits each predicate into its atoms and thus, produces the most fine-grained result. In contrast, the strategy *NONE* performs no additional operations resulting in the coarsest granularity of the predicates. In addition to these trivial strategies, there are our novel predicate split strategies *PDG*, $PDG_{ctrl}$ and *SSA* which split the predicates generated by the refinement procedure based on dependency information extracted from the program or control dependence graph or the difference in the SSA-maps accompanying the states in the counterexample trace. All experiments are performed for each of those strategies.

## 5.3 Sensibility Study

CPACHECKER is a highly configurable system and each configuration can potentially lead to different results of our experiments. Obviously, we cannot test all configurations. We can, however, study the influence of the configuration options we consider to be the most important. One such option is the *SMT-solver* that is used for the abstraction and refinement computations, as it might be that some of the split strategies produce formulas that one solver can handle better than another. The second option we consider very important is the *block operator*—the operator that determines when an abstraction should be computed. This operator has a huge influence on the block formula that needs to be over-approximated by the abstraction computation.

**Experiments** To empirically support our choice of configuration, we perform a sensibility study for the aforementioned configuration options. Therefore, we randomly selected 250 programs from the SVCOMP case study and created verification tasks using different solvers and block operators.

CPACHECKER currently supports four different SMT-solvers: MATHSAT 5, PRINCESS, SMTINTERPOL and Z3. However, only MATHSAT 5 and Z3 work for our verification tasks and therefore are used in this study. The other

[3] https://sv-comp.sosy-lab.org/2019/benchmarks.php

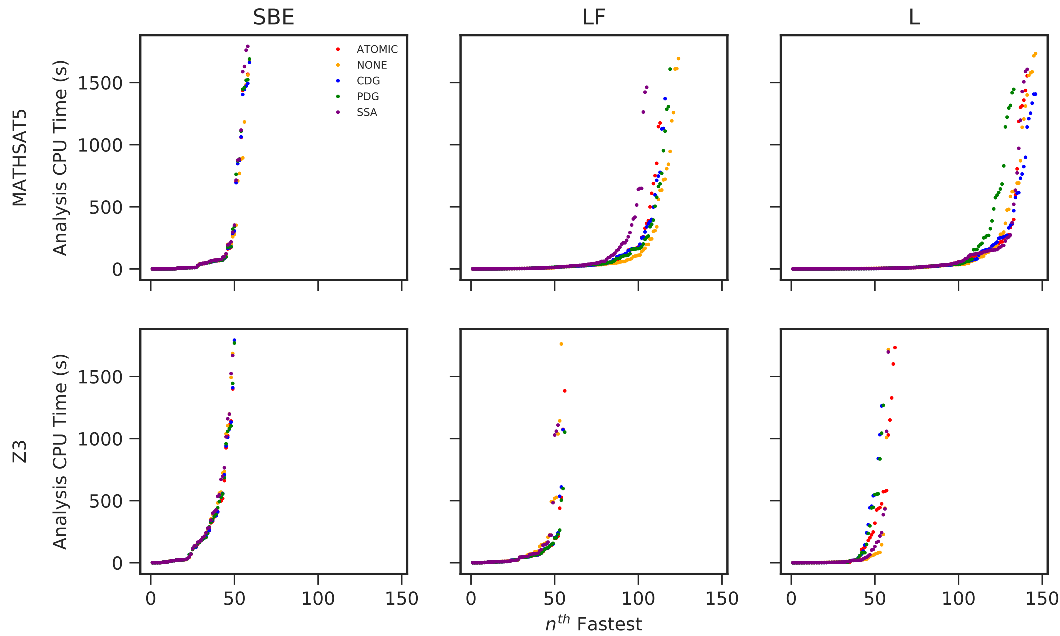



FIGURE 5.2: Quantile plots of the analysis CPU time for the results of the sensibility study.

solvers do not support the required theories (SMTINTERPOL cannot handle bit-vectors and princess does not support floats).

We explore three different block operators: The traditional way of summarizing program blocks is called single-block encoding (*SBE*). With this block operator abstractions are computed after every control-flow edge. However, this approach was discovered to be unfavorable as it explores a huge number of paths in the abstract state-space and it was eventually replaced by large block encoding [7] and its generalization adjustable block encoding [6]. Large block encoding is implemented in the block operator *L* with which abstractions are computed only at loop heads. The third block operator *LF* produces somewhat smaller blocks as abstractions are computed not only at every loop head but also at every function call. For all other parameters of CPACHECKER, the default values are used. Together with the five split strategies, this results in a total of 7500 verification tasks.

**Results** The results for this sensibility study are shown in figure 5.2. The figure shows quantile plots for the analysis CPU time grouped by configuration. One can clearly see that MATHSAT 5 outperforms Z3 in terms of the number of solved tasks with all block operators and split strategies. As for the block operators, as expected the configurations with *SBE* could solve the least tasks followed by *LF* and then with the most solved tasks *L*. As MATHSAT 5 in combination with *L* produces by far the most results, we decide to use this configuration for the rest of our experiments.

## 5.4 Results

We discuss the results of our experiments for each case study separately. As the main goal of this work is to understand the interaction between predicate granularity and predicate abstraction, we take a more detailed look at the case study SCENARIOS. The SVCOMP case study is there to see whether our findings generalize to a broader variety of programs.

### 5.4.1 Case Study SCENARIOS

First, we take a detailed look at the results for the case study SCENARIOS. Plots for the relevant metrics are shown in figure 5.3. The x-axis of the plots is labeled with the problem size. The first row of the figure contains plots for the analysis CPU time as reported by CPACHECKER measured in seconds and rounded to two decimal places. The analysis CPU time only measures the time spent for the actual analysis and excludes any setup times like parsing the program under verification or the creation of a program dependence graph. The second row shows the average number of models that were enumerated during abstraction computations. We use averages here, as there are many abstraction computations performed in each verification task and we are interested in an overall picture and not only in single abstraction computations. Also, this allows comparing different configurations of the same verification problem that took a different number of abstraction computations. In the third row, the average number of predicates used during abstraction computations are shown, and the last row depicts the number of refinements that were needed to solve the verification task. Note that the y-axes are plotted on a logarithmic scale. Data points for tasks that could not be solved within the given resource limits are omitted as most of those do not carry any meaningful information.

#### RQ1.1 Number of Models

In the scenario SEQUENTIAL, we observe an exponential growth in the number of models generated during abstraction when using the predicate split strategy *ATOMIC*. The reason is that each of the $2^n$ paths from the start of the program to the abstraction location (the head of the loop, see figure 5.1) can be expressed with the fine-grained predicates generated by this split strategy. For all other strategies, the number of models grows linearly with the problem size. The predicates generated by these split strategies already encode the relationships between the variables `ai` and `li` that are important for solving the verification task. Together with the way CPACHECKER traverses the program's control flow graph, [4] this results in a linear number of models enumerated during abstraction.

In the scenario REACTIVE, the number of models does not explode for any of the split strategies as it does in the scenario SEQUENTIAL. With the split

[4] By default, CPACHECKER uses a so-called *reverse-postorder* traversal strategy. This has the effect that else-branches are always taken first.

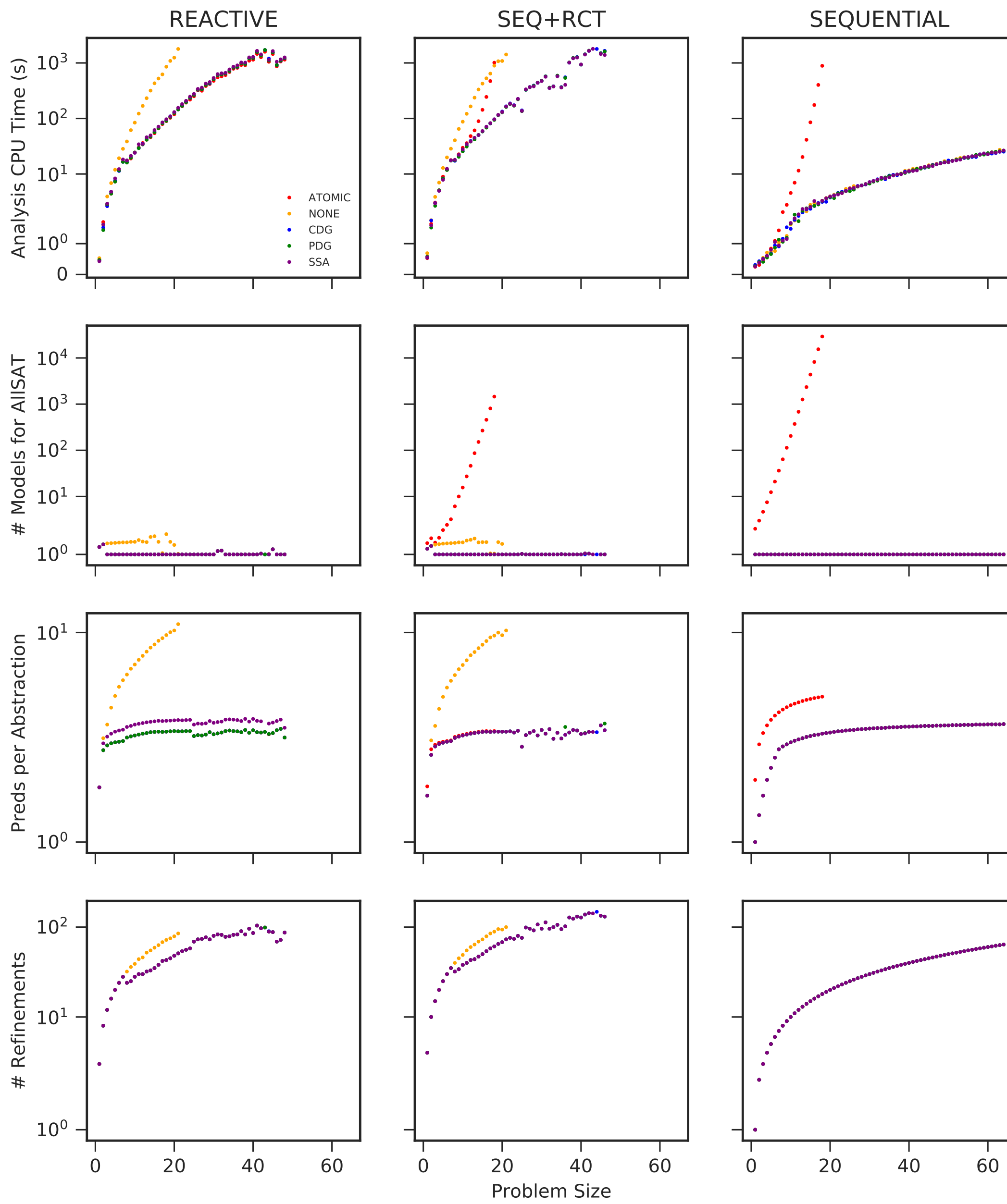



FIGURE 5.3: Results for the case study SCENARIOS grouped by scenario

strategy *NONE*, the number of models is slightly higher than it is for the other strategies. But that does not solely explain the huge difference in the running time of the tasks. Instead, we can observe that the average number of predicates used during the abstraction computations grows very quickly when using the split strategy *NONE*. At first sight, this might seem surprising as this strategy does not split any of the predicates generated during refinement into multiple new predicates. But having very fine-grained predicates can actually reduce their number as predicates typically have parts in common so splitting predicates into smaller parts or even atoms will most likely produce a lot of duplicates. The larger number of predicates also adds to the complexity of the abstraction problem as the abstraction formula grows bigger and more complex. The strategies *ATOMIC*, *PDG* and $PDG_{ctrl}$ behave very similarly to each other in this scenario. They all result in fewer models than the strategy *NONE* and the number of predicates grows slower.

With the explanations from the other two scenarios, the results for the scenario SEQ+RCT are as expected. As this scenario is the sequential composition of the other two, the observed behavior can be described informally as the *sum* of the observations of the other scenarios. For the split strategy *NONE*, the number of models explodes because of the SEQUENTIAL part of the scenario. Similar, the behavior with the strategy *NONE* is almost identical to the scenario REACTIVE. All of our novel split strategies *PDG*, $PDG_{ctrl}$ and *SSA* outperform the two trivial strategies concerning the number of models encountered during abstraction in this scenario.

Overall, all three of our novel split strategies produced the least number of models in all three scenarios. However, this metric alone proved to be not as good an indicator for the expected performance of predicate abstraction as we expected it to be. But more factors, like the number and structure of the predicates and the block formula, need to be considered.

### RQ1.2 Number of Refinements

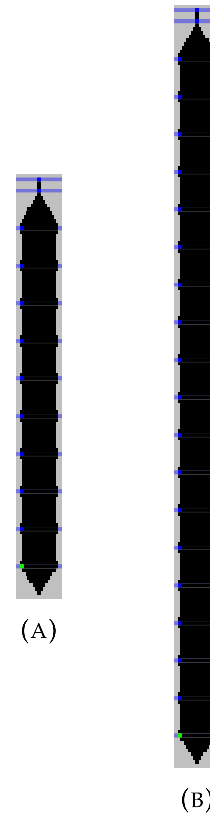
In terms of the number of refinements, there is no difference between the different split strategies for the scenario SEQUENTIAL. As for the scenarios REACTIVE and SEQ+RCT, the strategy *NONE* needs several refinements more than the other strategies for problem sizes greater than 7. This not very surprising as the coarse-grained predicates produced by this strategy take away more of the flexibility in over-approximating the block formula at hand the abstraction procedure has. There is almost no difference in refinements for the remaining four split strategies (the markers in the plots are invisible because they are on top of each other). That suggests that our novel split strategies find a good compromise in the granularity of predicates according to this metric.

### RQ2.1 Efficiency

For the predicate split strategy *ATOMIC*, one can see how the costs in terms of analysis time grow more than exponentially with the problem size of the scenario SEQUENTIAL. This is caused by the exponential number of models

the abstraction procedure has to deal with in this setting and demonstrates how the costs for the `AllSAT` query performed during the abstraction computation grow rapidly. All other split strategies avoid those costs because they keep the predicates more coarse-grained resulting in costs almost linear in $n$ solving even the largest problem instances in this scenario.

Scenario REACTIVE draws a different picture. This time, costs grow quickest for the split strategy *NONE*. As discussed in the section about the number of models encountered during abstraction, we explain these costs with the larger number and size of the abstraction predicates that add to the complexity of the abstraction problem. In addition, the size of the explored state space is larger as the abstraction procedure is not able to generalize over larger parts of the state-space resulting in more abstraction computations. Figure 5.4 shows graphical representations of the explored state space for this scenario with problem size $n = 5$ when using the predicate split strategy *ATOMIC* (figure 5.4a) and *NONE* (figure 5.4b). The split strategy *NONE* certainly explored more of the state-space.



(A)

(B)

FIGURE 5.4: ARG for scenario REACTIVE with split strategy *ATOMIC* (left) and *NONE* (right)

In the last scenario SEQ+RCT, the time needed for the analysis with each split strategy should be approximately the sum of the times of the previous two scenarios. Looking at the results this is essentially the case. As a result, all three of our novel predicate split strategies $PDG_{ctrl}$, *PDG* and *SSA* outperform the two trivial split strategies in terms of CPU time for the analysis very soon as the problem sizes grow, demonstrating that predicate abstraction can benefit from a well-chosen granularity of the abstraction predicates.

### RQ2.2 Effectiveness

For the scenario SEQUENTIAL, the least effective predicate split strategy is *ATOMIC*. Because of the exponential growth in running time, this strategy can solve only 18 of the 64 tasks. All other split strategies can solve all of the 64 tasks of this scenario.

For the scenario REACTIVE, the strategy *NONE* is the least effective split strategy with only 21 solved tasks. All other strategies can solve more than twice as many tasks. The strategies *ATOMIC*, *PDG*, and $PDG_{ctrl}$ are the most effective and manage to solve 48 tasks. The strategy *SSA* is almost as effective in this scenario solving 47 tasks.

In the last scenario SEQ+RCT, the trivial split strategies *ATOMIC* and *NONE*, solving only 18 or 21 tasks respectively, perform far worse than the other strategies. Here, the strategies $PDG_{ctrl}$ and *SSA* are the most effective

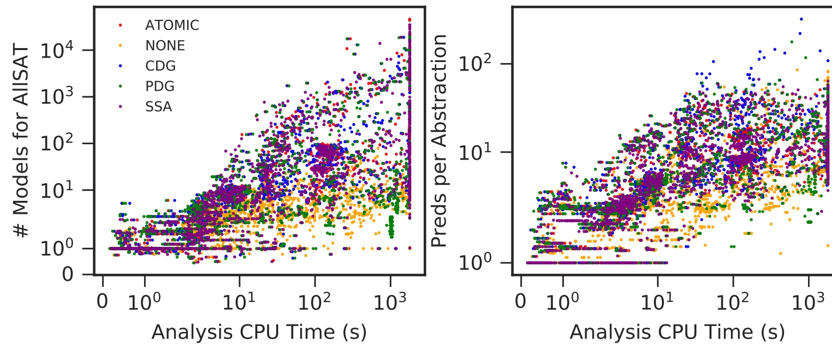



FIGURE 5.5: Scatter plots for the case study SVCOMP

with 45 solved tasks—more than twice as many as the trivial split strategies, followed by the strategy *PDG* which can solve 44 tasks.

Overall, it can be seen that, while the strategies *ATOMIC* and *NONE* perform good (in terms of number of solved tasks) in only one of the scenarios, all of our novel split strategies show the same effectiveness as the better of those strategies in the scenarios SEQUENTIAL and REACTIVE and even outperform both in the scenario SEQ+RCT. However, this case study does not show any significant difference in effectiveness between the strategies *PDG*, $PDG_{ctrl}$, and *SSA*.

### 5.4.2 Case Study SVCOMP

With the case study SVCOMP, we investigate whether our novel predicate split strategies also produce as good results as they do for the scenarios and whether we can observe the same effects. We excluded all tasks from the results that caused an exception with any of the used configurations or that could not be solved with any of the five predicate split strategies within the given resource limits. That means that for every task included in the results, for all split strategies, the result was one of `TRUE`, `FALSE` or `UNKNOWN` and at least one of the strategies came to a verdict `TRUE` or `FALSE`. The exceptions we encountered were caused either by unsupported features or by software bugs in CPACHECKER. In the end, there are a total of 1755 programs remaining for which we investigate the results.

#### RQ1.1 Number of Models

Figure 5.5 shows scatter plots for the average number of models encountered during abstraction and the analysis time, as well as the average number of predicates and the analysis time. It can be seen that for most of the tasks, the maximum number of models during an abstraction computation is comparatively low (note the logarithmic scale on both axes). On average, the least models occur with the predicate split strategy *NONE*. This is expected as this strategy produces the most specific predicates that do often apply only to a specific part of the path formula. All other four strategies produce a considerably higher number of models on average but behave similarly among

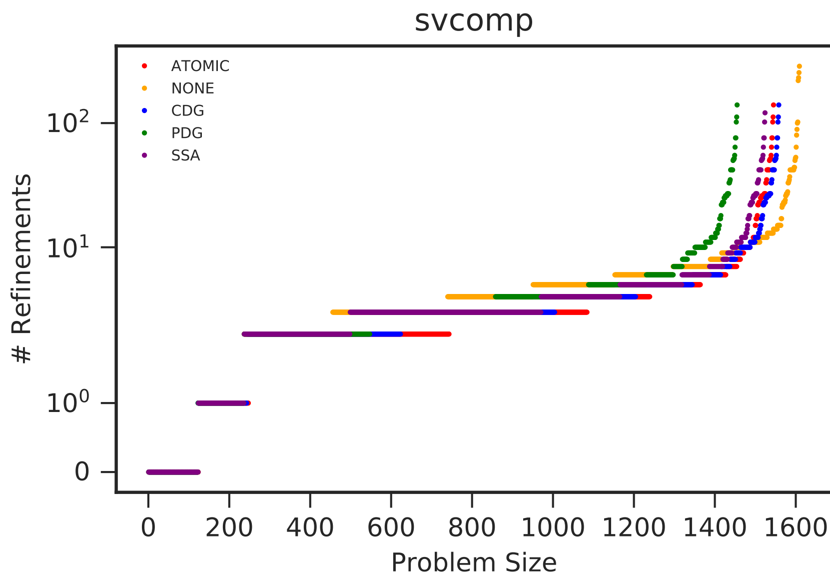



FIGURE 5.6: Quantile plot of the number of refinements for the case study SVCOMP

themselves. This is most prominent in the categories *ControlFlow*, *ProductLines*, and *Sequentialized*, all of which share some characteristics with the scenario SEQUENTIAL. While there is a tendency that more models mean a longer running time of the analysis, that does not hold for all tasks. There are many tasks where the number of models is small but the analysis still takes a long time to complete. This is most noticeable with the split strategy *NONE* where the number of models is low in general. Growth in the number of predicates, like discussed for the scenario REACTIVE, cannot always be observed in these cases. The number of predicates rather grows with the number of models. All in all, our novel predicate split strategies do not succeed very well in reducing the number of models that have to be enumerated during the predicate abstraction computations leaving potential for improvement.

### RQ1.2 Number of Refinements

The number of refinements does not vary very much between the different predicate split strategies. The strategies *NONE* and *PDG* tend to need some more refinements than the other strategies as they tend to produce the most coarse-grained predicates. In the case of the strategy *PDG*, this could also be an artifact of the reduced number of data points available for this strategy. As for the other three predicate split strategies, the number of refinements is very similar and most of the differences between these can be attributed to missing results. The fact that the strategies $PDG_{ctrl}$ and *SSA* are so similar to the strategy *ATOMIC* that produces the most fine-grained predicates possible, suggests that these strategies produce predicates of a granularity that leaves the abstraction procedure enough flexibility to generalize and avoid any additional refinement iterations.

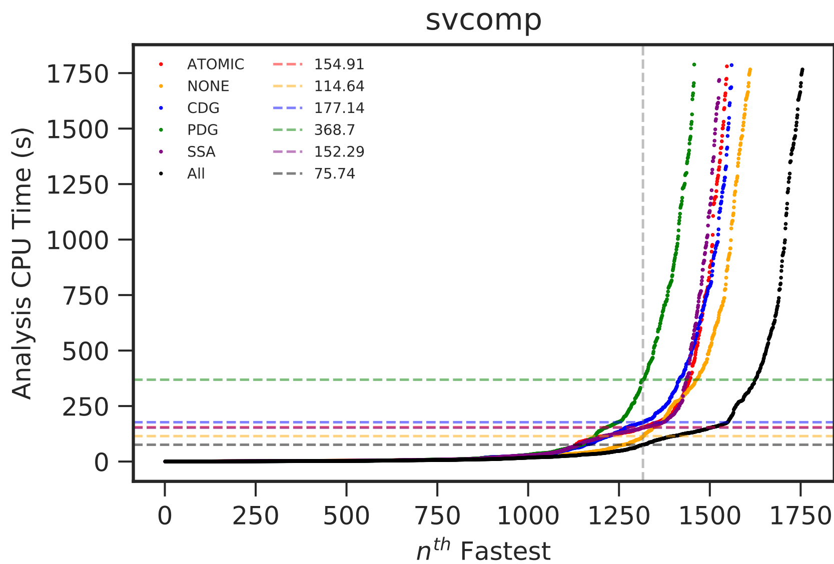



FIGURE 5.7: Quantile plot of the analysis CPU time for the SV-COMP case study.

### RQ2.1 Efficiency

Reasoning about the efficiency of the different split strategies in this case study is not easily possible. Theoretically, we would have to reason about efficiency on a per-program basis, but given the number of tasks in this study, this is not feasible. Instead, we look at the overall efficiency of the different split strategies and go into more detail only in interesting or unexpected cases. We consider a split strategy $A$ to be more efficient than another strategy $B$, if for every number $n$ of tasks, the $n^{th}$-fastest task solved by $A$ was solved faster than the $n^{th}$-fastest task solved by $B$. Or put differently, a split strategy $A$ is more efficient than a split strategy $B$ if given a time limit $t$, with strategy $A$ more tasks can be solved in under $t$ seconds than with strategy $B$. Quantile plots of the analysis times for this case study are shown in figure 5.7. The dashed lines mark the third quartile for each predicate split strategy with the color matching quantile plot. It can be seen that most of the tasks can be solved in under 200 seconds by almost all predicate split strategies, meaning that these tasks are comparatively easy to solve and therefore are less interesting in this study.

Conspicuously, the strategy *PDG* is noticeably less efficient than the others. The reason for this is that for some of the larger programs in the category *ECA*, the control dependence graph becomes very huge because these programs contain a considerable amount of control and data dependencies. As a result, slicing this huge PDG is very expensive. In some cases, the actual analysis does not even start because the time budget is completely spent creating the PDG. It should be noted that, as the PDG can be computed once for every program and then be reused, this counts as a pre-processing step and is not included in the analysis CPU time. But in these cases, a timeout can still be expected, even if the PDG creation would be instant, because of the

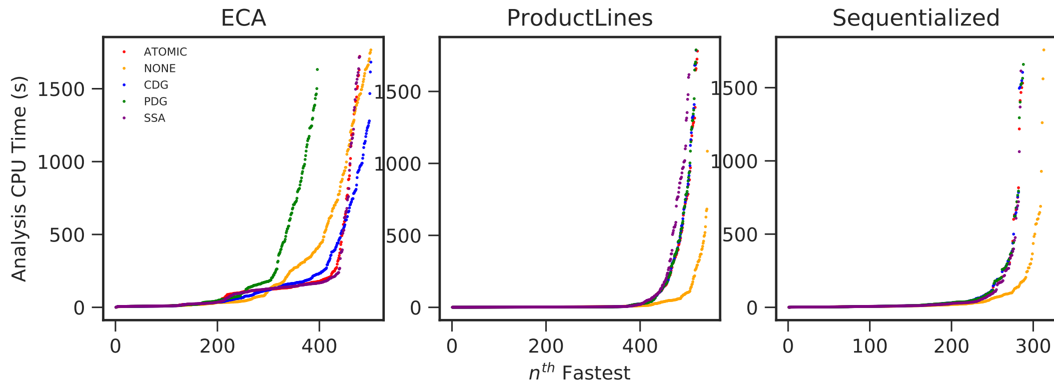



FIGURE 5.8: Quantile plots for selected categories of the SV-COMP case study.

high costs for program slicing. This was the only case where we observed the predicate splitting to cause any noticeable overhead with any of the predicate split strategies. In the vast majority of tasks, only a few seconds were spent for the predicate split operator. The similar split strategy $PDG_{ctrl}$ avoids this problem most of the time by not including any data dependencies. Only for a few of the tasks mentioned before does the split operator of this strategy cause any noticeable overhead. But in contrast to the split strategy *PDG*, these costs are not caused by slicing, but by the actual splitting procedure that traverses the interpolant formula and splits it into multiple predicates. The same observation can be made for the split strategy *SSA* but in even fewer cases. The costs for splitting are never a problem for the strategies *NONE* and *ATOMIC* as for the former no splitting is done at all and for the later splitting can be performed more efficiently.

According to our notion of efficiency, the strategy *NONE* is the most efficient overall. For almost any number of tasks, this strategy needed the least time to solve them. It is only superseded by the strategies *ATOMIC* and *SSA* for a small fraction of tasks. The cause is again several sets of tasks in the category *ECA* that could be solved much more efficiently with the strategies that produce more fine-grained predicates. As the programs in this category have a similar structure to our scenario REACTIVE, such behavior was expected. In the categories *ProductLines* and *Sequentialized*, the strategy *NONE* also was considerably more efficient than the others as can be seen in figure 5.8.

The strategies *ATOMIC* and *SSA* both behave very similar in terms of efficiency. Their quantile plots are almost identical for the fastest 1400 verification tasks. Afterwards, *ATOMIC* is slightly more efficient than *SSA* and both strategies are eventually superseded by the strategy $PDG_{ctrl}$. Once more, the difference mainly stems from the category *ECA* where $PDG_{ctrl}$ was more effective.

**RQ2.2 Effectiveness**

In contrast to our expectations, the most effective split strategy in this case study is not one of our novel split strategies but the strategy *NONE* that

| Category | Tasks | # Solved Tasks | | | | |
|---|---|---|---|---|---|---|
| | | ATOMIC | NONE | CDG | PDG | SSA |
| Arrays | 40 | 38 | 37 | 38 | 38 | 37 |
| BitVectors | 37 | 34 | 36 | 33 | 34 | 35 |
| ControlFlow | 76 | 76 | 76 | 76 | 76 | 76 |
| ECA | 612 | 479 | 501 | 501 | 396 | 479 |
| Loops | 111 | 111 | 105 | 106 | 106 | 110 |
| ProductLines | 554 | 522 | 543 | 519 | 519 | 504 |
| Sequentialized | 325 | 287 | 313 | 287 | 288 | 285 |
| **All** | **1755** | **1547** | **1611** | **1560** | **1457** | **1526** |

TABLE 5.1: Number of solved tasks by category and split strategy.

could solve 1611 of the 1755 tasks. The only categories where another strategy was more effective are the categories *Arrays* and *Loops* where this strategy performed worst. For the latter category, this is not surprising, as the coarse-grained predicates of this strategy hinder the abstraction procedure in finding loop invariants. Special focus must also be laid on the category *ECA* which stands for *event-condition-action* and therefore contains programs with a structure similar to our REACTIVE scenario. Strangely, the strategy *NONE* which performed worst in the scenario REACTIVE is the most effective strategy in this category. This is mainly due to the fact that there are two groups of very large programs in this category where this strategy was the only these tasks could be solved with. Unfortunately, we could not identify the reason behind this behavior as there is no obvious difference to other programs in this category (where *NONE* is often less efficient and effective) besides the larger size of these programs. However, the size of the programs alone cannot explain why this split strategy was able to solve these tasks when it could not solve tasks with similar-looking but smaller programs. This issue needs further investigation and is postponed to future work.

## 5.5 Discussion

In the case study SCENARIOS, we saw that the granularity of predicates can have a tremendous impact on the efficiency and effectiveness of predicate abstraction and thus on the analysis. And while our novel split strategies demonstrated that it is possible to handle two very different extreme cases at once, the results for the case study SVCOMP are rather sobering. The most promising strategy is $PDG_{ctrl}$ that is more effective than CPACHECKER's default configuration that uses the strategy *ATOMIC*. But it is still less efficient and less effective than the split strategy *NONE* that was also available prior to this work. However, this also shows that the current default strategy for deciding the predicate granularity in CPACHECKER is not necessarily the optimal choice.

Despite the bad performance of our novel predicate split strategies, the results of our experiments confirm that the granularity of predicates is an

important factor for the performance of `AllSAT`-based Boolean predicate abstraction. While we have not yet found a suitable predicate split strategy for a wider variety of programs, we can construct one additional strategy that would outperform all of the current five predicate split strategies: The black line in figure 5.7 plots the analysis times theoretically achievable if the analysis would be run for each of the five split strategies in parallel until the first one terminates. It can be easily seen that this strategy would be the most efficient and also the most effective. This line is not identical to the minimum of the two trivial strategies, showing that our novel split strategies indeed perform better than those in some cases.

## 5.6 Threats to Validity

In this section, we discuss several aspects of the design and implementation of the experiments in this chapter that put a threat to the validity of its results.

### 5.6.1 Internal Validity

As CPACHECKER is written in Java, one must be aware of characteristics of the JVM like the garbage collector or "warm-up" effects of the JIT-compiler, which can influence the benchmark results. However, as we do not rely on very precise measurements but either look at time-independent numbers or at time spans where such effects can be neglected, JVM-related issues are not much of a concern.

What cannot be neglected, however, are programming bugs in CPACHECKER or our implementation. A recent study discovered several such bugs [26] and most likely not all of them are fixed in our implementation. After inspection of the logs of some of the tasks that terminated with an exception, we are confident that some of those exceptions were caused by programming bugs. While this may distort the overall picture (results for these tasks are missing), these tasks do not introduce incorrect results. Also, some tasks produced incorrect results, however, these errors do not depend on the selected split strategy.

### 5.6.2 External Validity

The selection of case studies plays an important role in the external validity of our experiments. The case study SCENARIOS is there to demonstrate extreme cases and does not allow for generalization. Therefore, we have the case study SVCOMP which is widely used in the software verification community. While this case study contains a variety of programs with different characteristics, those are not equally distributed. Most of the programs are part of the categories *ECA*, *ProductLines* and *Sequentialized* biasing the results towards the characteristics of these programs.

Another important aspect is the chosen configuration. CPACHECKER is a highly configurable system and each configuration option potentially influences the results of our study. For the options we deem the most important, we performed a sensibility study and chose the options that showed the best performance. For the other configuration options, we used CPACHECKER's default values assuming that those are chosen reasonably by the developers.

# Chapter 6

# Related Work

In chapter 4, we already mentioned path slicing [20]. Path slicing was created in order to reduce the size of counterexample traces in order to make manual or automatic inspection thereof simpler. Path slicing indirectly influences the granularity of abstraction predicates in that it modifies the counterexample trace, what on the other hand has an effect on what predicates are generated by the refinement procedure. The slicing technique used in some of our predicate split strategies was inspired by path slicing but is not equivalent (see chapter 4).

In [17], Gurfinkel et al. present a different technique for predicate abstraction that avoids the exponential costs of AllSAT by relying on linear decision diagrams (LDDs)[8]. It would be interesting to investigate the role of predicate granularity with this approach. However, this is out of the scope of this work. In addition to the LDD-based predicate abstraction technique, they also introduce a different way to encode program blocks that separates control- and data-flows and preserves the control-flow structure. As our predicate split strategies also separate control-flow information, it is possible that an AllSAT-based predicate abstraction in combination with these split strategies could benefit from such a block encoding.

In the results of chapter 5, we discussed that an approach that runs several predicate split strategies in parallel would theoretically achieve better results than each strategy in isolation. This leads to the question if it is possible to select a strategy for each task separately in advance. There are several works that use different approaches to select configurations by analyzing the input program of a verification task. Apel et al. [2] analyze usage patterns of program variables in order to assign them to appropriate abstract domains. Demyanova et al. [13] use a data-flow analysis in order to classify the roles of program variables and predict the categories of the SVCOMP program set the different programs belong to. This work is extended in [14] where the variable roles and other empirical metrics of the program under verification are used to learn a model that predicts the performance of a verification tool or configuration. Such techniques could be adapted to select a fitting predicate split strategy.

Leroux et al. introduced a formalism for abstracting the interpolation problem in the refinement step of CEGAR, called *interpolation abstraction* [23]. In this approach, domain-specific knowledge provided by the user is used to guide a search in the lattice of possible interpolants to find better interpolants. This is an example of a different problem that fits into our concept

of an abstraction problem. Here, the underlying problem is to find good interpolants. Interpolation abstraction restricts the interpolation procedure to those interpolants that can be constructed using the given abstraction precision, i.e., the additional information provided by the user. Also, the interpolants found by this technique may contain predicates of a different, better, granularity.

# Chapter 7

# Summary

In this work, we investigated the problem that the costs for predicate abstraction can explode when the abstraction predicates are chosen disadvantageously. Therefore, we first characterized this problem as an *abstraction problem*, the problem of finding an appropriate abstraction for the input of some underlying task, in this case, a verification task, such that this task can be solved more efficiently. We discovered that the *granularity* of the abstraction predicates, that is, their size and structure, have an influence on the performance of the predicate abstraction procedure. On the one hand, the granularity of predicates must not be too fine, as this can lead to more expensive abstraction computations due to an increased number of models that must be enumerated by the abstraction procedure. On the other hand, a too coarse granularity can make predicates unnecessarily complex and specific, and thus, limit the abstraction procedure's flexibility. This problem was demonstrated with a set of scenarios where neither very fine-grained predicates, nor coarse-grained predicates are a suitable choice.

In order to explore other levels of granularity, we extended the refinement procedure of CEGAR with an additional operator that extracts predicates of the desired granularity from the interpolants. We developed several strategies on how to split the interpolants appropriately by using information about control- and data-flow dependencies of the program under verification, as such dependency information is considered to be necessary to solve the verification task at hand. With these strategies, the granularity of predicates better suits the abstraction problems in these scenarios and as a result, the verification tasks can be solved much more efficiently. However, experiments with a larger variety of programs showed that none of the predicate split strategies, neither one of our novel strategies, nor one of the trivial strategies, is the single best choice. Instead, the different split strategies work best for different programs with different characteristics.

A task for future work is to refine the predicate split strategies presented in this work into yet another strategy that works with a wider variety of programs and outperforms all of the currently available strategies. Therefore, additional insights in the relationship between predicate granularity and abstraction costs are needed as this work showed that the number of models of the abstraction formula while being a relevant factor, is not the sole reason for high abstraction costs.

# Bibliography


[1] T. Agerwala and Jayadev Misra. *Assertion Graphs for Verifying and Synthesizing Programs*. Tech. rep. Austin, TX, USA, 1978.

[2] Sven Apel et al. "Domain Types: Abstract-Domain Selection Based on Variable Usage". In: *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*. 2013, pp. 262–278. DOI: 10.1007/978-3-319-03077-7_18. URL: https://doi.org/10.1007/978-3-319-03077-7%5C_18.

[3] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. "Boolean and Cartesian Abstraction for Model Checking C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*. 2001, pp. 268–283. DOI: 10.1007/3-540-45319-9_19. URL: https://doi.org/10.1007/3-540-45319-9%5C_19.

[4] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis". In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. 2007, pp. 504–518. DOI: 10.1007/978-3-540-73368-3_51. URL: https://doi.org/10.1007/978-3-540-73368-3%5C_51.

[5] Dirk Beyer and M. Erkan Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 2011, pp. 184–190. DOI: 10.1007/978-3-642-22110-1_16. URL: https://doi.org/10.1007/978-3-642-22110-1%5C_16.

[6] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. "Predicate abstraction with adjustable-block encoding". In: *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*. 2010, pp. 189–197. URL: http://ieeexplore.ieee.org/document/5770949/.

[7] Dirk Beyer et al. "Software model checking via large-block encoding". In: *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*. 2009, pp. 25–32. DOI: 10.1109/FMCAD.2009.5351147.

[8] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. “Decision diagrams for linear arithmetic”. In: *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*. 2009, pp. 53–60. DOI: 10.1109/FMCAD.2009.5351143.

[9] Edmund M. Clarke et al. “Counterexample-Guided Abstraction Refinement”. In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. 2000, pp. 154–169. DOI: 10.1007/10722167_15. URL: https://doi.org/10.1007/10722167%5C_15.

[10] Patrick Cousot and Radhia 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, California: ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[11] William Craig. “Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem”. In: *J. Symb. Log.* 22.3 (1957), pp. 250–268. DOI: 10.2307/2963593.

[12] Ron Cytron et al. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”. In: *ACM Trans. Program. Lang. Syst.* 13.4 (1991), pp. 451–490. DOI: 10.1145/115372.115320.

[13] Yulia Demyanova, Helmut Veith, and Florian Zuleger. “On the concept of variable roles and its use in software analysis”. In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. 2013, pp. 226–230. URL: http://ieeexplore.ieee.org/document/6679414/.

[14] Yulia Demyanova et al. “Empirical software metrics for benchmarking of verification tools”. In: *Formal Methods in System Design* 50.2-3 (Jan. 2017), pp. 289–316. DOI: 10.1007/s10703-016-0264-5.

[15] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The Program Dependence Graph and Its Use in Optimization”. In: *ACM Trans. Program. Lang. Syst.* 9.3 (1987), pp. 319–349. DOI: 10.1145/24039.24041.

[16] Susanne Graf and Hassen Sadi. “Construction of Abstract State Graphs with PVS”. In: *Computer Aided Verification, 9th International Conference, CAV ’97, Haifa, Israel, June 22-25, 1997, Proceedings*. 1997, pp. 72–83. DOI: 10.1007/3-540-63166-6_10. URL: https://doi.org/10.1007/3-540-63166-6%5C_10.

[17] Arie Gurfinkel, Sagar Chaki, and Samir Sapra. “Efficient Predicate Abstraction of Program Summaries”. In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. 2011, pp. 131–145. DOI: 10.1007/978-3-642-20398-5_11. URL: https://doi.org/10.1007/978-3-642-20398-5%5C_11.

[18] Thomas A. Henzinger et al. “Abstractions from proofs”. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. 2004, pp. 232–244. DOI: 10.1145/964001.964021.

[19] Susan Horwitz, Thomas W. Reps, and David W. Binkley. “Interprocedural Slicing Using Dependence Graphs”. In: *ACM Trans. Program. Lang. Syst.* 12.1 (1990), pp. 26–60. DOI: 10.1145/77606.77608.

[20] Ranjit Jhala and Rupak Majumdar. “Path slicing”. In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. 2005, pp. 38–47. DOI: 10.1145/1065010.1065016.

[21] Ranjit Jhala and Rupak Majumdar. “Software Model Checking”. In: *ACM Comput. Surv.* 41.4 (Oct. 2009), 21:1–21:54. ISSN: 0360-0300. DOI: 10.1145/1592434.1592438.

[22] Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. “SMT Techniques for Fast Predicate Abstraction”. In: *Computer Aided Verification*. Ed. by Thomas Ball and Robert B. Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 424–437. ISBN: 978-3-540-37411-4. DOI: 10.1007/11817963_39.

[23] Jérôme Leroux, Philipp Rümmer, and Pavle Subotic. “Guiding Craig interpolation with domain-specific abstractions”. In: *Acta Inf.* 53.4 (2016), pp. 387–424. DOI: 10.1007/s00236-015-0236-z.

[24] Jens Meinicke et al. “On essential configuration complexity: measuring interactions in highly-configurable systems”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. ACM Press, 2016. DOI: 10.1145/2970276.2970322.

[25] Mark Weiser. “Program Slicing”. In: *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981.* 1981, pp. 439–449. DOI: 10.1109/tse.1984.5010248. URL: http://dl.acm.org/citation.cfm?id=802557.

[26] Chengyu Zhang et al. “Finding and understanding bugs in software model checkers”. In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019.* 2019, pp. 763–773. DOI: 10.1145/3338906.3338932.

**Eigenständigkeitserklärung:**

Hiermit bestätige ich Sebastian Böhm, dass ich die vorliegende Arbeit selbstständig und ohne unzulässige Hilfe verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die wörtlich und sinngemäß übernommenen Passagen aus anderen Werken kenntlich gemacht habe. Die Arbeit ist weder von mir noch von einer anderen Person an der Universität Passau oder an einer anderen Hochschule zur Erlangung eines akademischen Grades bereits eingereicht worden.

Passau, den 18. September 2019 ____________________________

Sebastian Böhm