University of Passau
Faculty of Computer Science and Mathematics
Chair for Software Systems

Dissertation

Effective Approaches to Abstraction Refinement for Automatic Software Verification

Stefan Löwe

April 28, 2017

Supervisor:	Prof. Dr. rer. nat. Dirk Beyer
External Examiner:	Prof. Dr. Jan Strejcek

Abstract

This thesis presents various techniques that aim at enabling more effective and more efficient approaches for automatic software verification.

After a brief motivation why automatic software verification is getting ever more relevant, we continue with detailing the formalism used in this thesis and on the concepts it is built on.

We then describe the design and implementation of the value analysis, an analysis for automatic software verification that tracks state information concretely. From a thorough evaluation based on well over 4 000 verification tasks from the latest edition of the International Competition on Software Verification (SV-COMP), we learn that this plain value analysis leads to an efficient verification process for many verification tasks, but at the same time, fails to solve other verification tasks due to state-space explosion. From this insight we infer that some form of abstraction technique must be added to the value analysis in order to also allow the successful verification of large and complex verification tasks.

As a solution, we propose to incorporate counterexample-guided abstraction refinement (CEGAR) and interpolation into the value domain. To this end, we design a novel interpolation procedure, that extracts from infeasible counterexamples interpolants for the value domain, allowing to form a precision strong enough to exclude these infeasible counterexamples, and to make progress in the CEGAR loop. We then describe several optimizations and extensions to these concepts, such that the value analysis with CEGAR becomes competitive for automatic software verification.

As the next step, we combine the value analysis with CEGAR with a predicate analysis, to obtain a more precise and efficient composite analysis based on CEGAR. This composite analysis is indeed on a par with the world's leading software verification tools, as witnessed by the results of SV-COMP'13 where this approach achieved the 2$^{\text{nd}}$ place in the overall ranking.

After having available competitive CEGAR-based analyses for the value domain, the predicate domain, and the combination thereof, we then turn our attention to techniques that have the goal to make all these CEGAR-based approaches more successful. Our first novel idea in this regard is based on the concept of infeasible sliced prefixes, which allow the computation of different precisions from a single infeasible counterexample. This adds choice to the CEGAR loop, while without this enhancement, no choice for a specific precision, i. e., a specific refinement, is possible.

In our evaluation we show, for both the value analysis and the predicate analysis, that choosing different infeasible sliced prefixes during the refinement step leads to major differences in verification effectiveness and verification efficiency.

Extending on the concept of infeasible sliced prefixes, we define several heuristics in order to precisely select a single refinement from a set of possible refinements. We make this new concept, which we refer to as guided refinement selection, available to both the value and predicate analysis, and in a large-scale evaluation we try to answer the question which selection technique leads to well suited abstractions and thus, to a more effective verification process. Additionally, we present the idea of inter-analysis refinement selection, where the refinement component of a composite analysis may decide which of its component analyses is best to be refined, and in yet another evaluation we highlight the positive effects of this technique.

Finally, we present the results of SV-COMP'16, where the verifier we contributed and which is based on the concepts and ideas presented in this thesis achieved the 1st place in the category `DeviceDriversLinux64`.

Acknowledgements

Writing this dissertation took six long years, and it was not always a joyful endeavor, with some letdowns and disappointments along the way. I could only accomplish this task with the help and encouragement of several people, and I want to express my deep appreciation to them.

First and foremost I would like to thank my supervisor Dirk Beyer for his untiring support, for the many fruitful discussions we had, and for sharing his knowledge and experience with me. I especially want to thank Dirk for bringing the International Competition on Software Verification to life.

I also want to express my gratitude to Jan Strejcek for reviewing my dissertation, as well as not holding it against me that I planned but never followed through with my visit in Brno.

I am also highly thankful to Philipp Wendler. I had the luxury to share the office with Philipp, and profit from his knowledge and his experience on a daily basis. The amount of work he puts into maintaining the CPAchecker project is exceptional, and besides that he always found time to fix our ageing computer infrastructure. On top of that, we had some joint papers and projects together all of which came to a successful conclusion in the end.

Furthermore, I want to thank my office colleagues Gregor Endler, Matthias Dangl, Karlheinz Friedberger, Peter Häring, Malte Rosenthal, and Andreas Stahlbauer for our joint work on papers and the CPAchecker project, as well as for the good times we had together over the years.

I also want to thank Sven Apel for making fun of me every once in a while, and even more for continuously motivating me and teaching me a lot of things.

Many thanks also to our assistants Eva Veitweber and Eva Reichhart, who always took care of all the administrative work without much fuss. And it was a pleasure to bring you coffee in exchange for a good laugh.

I also want to thank the student assistants that worked hard on the CPAchecker framework and the benchmarking infrastructure, namely Alexander Driemeyer, Thomas Lemberger, Sebastian Ott, and Thomas Stieglmaier.

Last but not least, I want to thank my friends and my whole family, especially my parents Christina and Helmut, my sister and my brothers, and of course my wonderful wife Katarzyna, who always were and will be there for me to support me.

Contents

1 Introduction **1**

1.1 The Need For Software Verification 1

1.2 Automatic Software Verification 2

1.3 Contributions 3

1.3.1 Value Analysis with CEGAR and Interpolation 5

1.3.2 Precise and Efficient Composite Analysis based on CEGAR . 5

1.3.3 Refinements over Infeasible Sliced Prefixes 6

1.3.4 Guided Refinement Selection 6

1.3.5 Contribution to SV-COMP'16 7

1.3.6 Availability of Implementations and Experimental Data . . . 7

1.4 Structure of the Thesis 7

2 Background **9**

2.1 Programs, Control-Flow Automaton, and Semantics 9

2.2 Configurable Program Analysis 10

2.2.1 Abstract Domain 10

2.2.2 Precision 11

2.2.3 Transfer Relation 11

2.2.4 Merge Operator 11

2.2.5 Stop Operator 12

2.2.6 Precision-Adjustment Operator 12

2.3 CPA Algorithm 12

2.4 Abstract Reachability Graph 14

2.5 Counterexample-Guided Abstraction Refinement 15

2.6 Interpolation 17

2.7 CPAchecker as Verification Framework 17

3 Value Analysis 21

3.1 Motivation 21

3.2 Related Work 24

3.3 Definitions 25

3.4 Value Analysis as CPA 27

3.4.1 Abstract Domain 27

3.4.2 Precision 27

3.4.3 Transfer Relation 28

3.4.4 Merge Operator 28

3.4.5 Stop Operator 28

3.4.6 Precision-Adjustment Operator 28

3.5 Evaluation 28

3.5.1 Setup 29

3.5.2 Benchmarks 29

3.5.3 Configuration 30

3.5.4 Results 30

3.6 Conclusion 33

3.6.1 Lessons Learned 33

3.6.2 Challenge 34

3.6.3 Proposition 34

3.6.4 Solution 34

4 Value Analysis with CEGAR and Interpolation 35

4.1 Motivation 35

4.2 Related Work 38

4.3 State-Space Exploration Algorithm for the Value Domain 39

4.4 Precision for the Value Domain 39

4.5 Feasibility Check for the Value Domain 40

4.6 Interpolation for the Value Domain 41

4.6.1 Interpolation for Abstract Variable Assignments 41

4.6.2 Interpolation for Constraint Sequences 42
4.7 Refinement Based on Value Interpolation 43
4.8 Evaluation 46
4.8.1 Configuration 46
4.8.2 Results 46
4.8.3 Comparison to the Plain Value Analysis 48
4.9 Conclusion 52
4.9.1 Lessons Learned 52
4.9.2 Challenge 52
4.9.3 Proposition 52
4.9.4 Solution 53

5 Value Analysis with Improved CEGAR and Interpolation 55
5.1 Motivation 55
5.2 Reducing the Number of Value Interpolation Queries 55
5.2.1 Iterative, Inductive Interpolation 56
5.2.2 Interpolation over Deepest Infeasible Suffix 56
5.2.3 Interpolant-Equality Heuristic 57
5.2.4 Interpolant-Equivalence Heuristic 58
5.2.5 Evaluation of the Optimizations for the Value Interpolation . 58
5.3 Reducing the Number of Refinements 59
5.4 Evaluation 60
5.4.1 Configuration 61
5.4.2 Results 61
5.4.3 Comparison to the Plain Value Analysis 62
5.4.4 Level of Non-Determinism 63
5.5 Versatility of Value-Analysis Refinement 68
5.5.1 Applicability to other Analyses 68
5.5.2 Regression Verification 69
5.6 Further Considerations 71
5.6.1 Static Refinement 71
5.6.2 Global Refinement 73

5.6.3 Impact-Like Refinement for the Value Analysis 76

5.7 Conclusion 78

5.7.1 Lessons Learned 78

5.7.2 Challenge 79

5.7.3 Proposition 79

5.7.4 Solution 79

6 Precise and Efficient Composite Analysis based on CEGAR 81

6.1 Motivation 81

6.2 Related Work 82

6.3 Composition of a Value Analysis and a Predicate Analysis 83

6.4 Evaluation 84

6.4.1 Configuration 84

6.4.2 Results 85

6.5 International Competition on Software Verification 2013 87

6.6 Conclusion 89

6.6.1 Lessons Learned 89

6.6.2 Challenge 89

6.6.3 Proposition 89

6.6.4 Solution 89

7 Refinements over Infeasible Sliced Prefixes 91

7.1 Motivation 91

7.2 Related Work 93

7.3 Introducing Infeasible Sliced Prefixes 94

7.4 Extracting Infeasible Sliced Prefixes 95

7.5 Refinements over Infeasible Sliced Prefixes 98

7.6 Evaluation 99

7.6.1 Infeasible Sliced Prefixes for the Value Analysis 100

7.6.2 Infeasible Sliced Prefixes for the Predicate Analysis 103

7.6.3 Infeasible Sliced Prefixes with Large-Block Encoding 106

7.6.4 Further Applications of Infeasible Sliced Prefixes 111

7.7 Conclusion 112
7.7.1 Lessons Learned 112
7.7.2 Challenge 112
7.7.3 Proposition 113
7.7.4 Solution 113

8 Guided Refinement Selection 115

8.1 Motivation 115
8.2 Related Work 116
8.3 Heuristics for Guided Refinement Selection 116
8.3.1 Selection by Domain-Type Score of Path Precision 117
8.3.2 Selection by Depth of Pivot Location of Path Precision . . . 117
8.3.3 Selection by Width of Path Precision 118
8.3.4 Selection by Length of Infeasible Sliced Prefix 119
8.3.5 Composition of Heuristics 119
8.3.6 Tailor-Made Heuristics using Domain Knowledge 119
8.4 Evaluation of Intra-Analysis Refinement Selection 120
8.4.1 Configuration 120
8.4.2 Refinement Selection for the Predicate Analysis 121
8.4.3 Refinement Selection for the Value Analysis 125
8.5 Refinement Selection for Composite Analyses 127
8.6 Evaluation of Inter-Analysis Refinement Selection 130
8.6.1 Configuration 131
8.6.2 Results 131
8.7 Conclusion 132
8.7.1 Lessons Learned 132
8.7.2 Challenge 132
8.7.3 Proposition 133

9 Contribution to SV-COMP'16 135

9.1 Configuration 135
9.2 Results of SV-COMP'16 and beyond 136

10 Summary and Future Research **141**

10.1 Summary 141

10.2 Future Research 143

List of Figures

1.1 Overview of the contributions made in this thesis 4

2.1 Overview of the architecture of the CPAchecker framework 18

3.1 Simple verification task, with its corresponding CFA 22
3.2 ARG of a simple verification task 23
3.3 Quantile plot for the value analysis 33

4.1 Simple verification task, containing an unbounded loop, with its corresponding CFA 36
4.2 ARG of a simple verification task, containing an unbounded loop . . 37
4.3 Examples of a CFA, an infeasible error path, a sequence of interpolants, and the respective precision elements 45
4.4 Scatter plot comparing the CPU times of the plain value analysis and the value analysis with CEGAR, with regard to the number of refinements 50
4.5 Quantile plot comparing the CPU times of the plain value analysis and the value analysis with CEGAR 51

5.1 Quantile plot comparing the CPU times of the plain value analysis, the value analysis with CEGAR, and the value analysis with improved CEGAR 64
5.2 Scatter plot comparing the CPU time of the plain value analysis and the value analysis with CEGAR for verification tasks with a lower level of non-determinism 66

5.3 Scatter plot comparing the CPU time of the plain value analysis and the value analysis with CEGAR for verification tasks with a higher level of non-determinism 67
5.4 A simple verification task with an interpolant tree 75

6.1 Example verification task exposing non-determinism due to missing initializer 81
6.2 Score-based quantile plot comparing the value analysis with improved CEGAR, the default predicate analysis of CPAchecker, and the composition of these two analyses 88

7.1 A verification task, an infeasible error path, and a “bad” and a “good” interpolant sequence 92
7.2 An infeasible error path, a cascade of sliced prefixes, and a set of infeasible sliced prefixes 97
7.3 Scatter plots comparing the CPU time of the value analysis using different heuristics for selecting infeasible sliced prefixes 102
7.4 Scatter plots comparing the CPU time of the predicate analysis using different heuristics for selecting infeasible sliced prefixes 105
7.5 Scatter plots comparing the CPU time of the predicate analysis if using the selection heuristic Length-Max and not explicitly using a selection heuristic for different ABE-block sizes 108
7.6 Visualization of extracting infeasible sliced (ABE) blocks 110

8.1 Quantile plot comparing the CPU time of predicate analysis without and with refinement selection using different heuristics 123
8.2 Quantile plot comparing the CPU time of value analysis without and with refinement selection using different heuristics 126
8.3 An example verification task, an infeasible error path, and a “bad” and a “good” interpolant sequence, the latter not being applicable to the value analysis 128
8.4 Visualization of inter-analysis refinement selection 129

9.1 Quantile plot comparing for several verifiers the number of correctly solved verification tasks for category DeviceDriversLinux64 137
9.2 Quantile plot comparing for several verifiers the number of correctly solved verification tasks from category IntegersControlFlow . . . 138

List of Tables

3.1 Overview of the verification tasks from SVCOMP'16 as used in this evaluation . . . 29
3.2 Table showing the verification effectiveness of the value analysis . . . 31
3.3 Table showing the verification efficiency of the value analysis 32

4.1 Table showing the verification effectiveness of the value analysis with CEGAR and interpolation . . . 47
4.2 Table showing the difference in verification effectiveness between the plain value analysis and the value analysis with CEGAR 49

5.1 Overview of the different optimization techniques and how they relate to not applying any optimizations and applying all optimizations . . 59
5.2 Table showing the verification effectiveness of the value analysis with improved CEGAR . . . 62
5.3 Table showing the difference in verification effectiveness between the plain value analysis and the value analysis with improved CEGAR . 63

6.1 Table showing the verification effectiveness of the composition of the value and predicate analysis . . . 85
6.2 Table showing the verification effectiveness of the composition of the value and predicate analysis in comparison to the value analysis with improved CEGAR . . . 86

7.1 Table showing the effects of two different selection heuristics for the value analysis . . . 101

7.2 Table showing the effects of two different selection heuristics for the predicate analysis . . . 104

8.1 Number of solved verification tasks for the predicate analysis with refinement selection using different heuristics . . . 122
8.2 Number of solved verification tasks for the value analysis with refinement selection using different heuristics . . . 125
8.3 Number of solved verification tasks for the three different analyses . 131

List of Algorithms

1 CPA($\mathbb{D}, R_0, W_0$) (taken from [26]) 13

2 CEGAR($\mathbb{D}, (e_0, \pi_0)$) (taken from [30]) 15

3 Interpolate(γ^-, γ^+) 43

4 Refine(σ) 44

5 ObtainInUseFunction(σ) 72

6 ExtractSlicedPrefixes(σ) 95

7 Refine$^+$(σ) 99

8 ExtractSlicedBlockPrefixes(σ) 107

List of Acronyms

ABE adjustable-block encoding

ARG abstract reachability graph

CEGAR counterexample-guided abstraction refinement

CFA control-flow automaton

CPA configurable program analysis

LBE large-block encoding

SAT satisfiability theories

SBE single-block encoding

SMT satisfiability modulo theories

SV-COMP International Competition on Software Verification

1 Introduction

In the first chapter we motivate why software verification is needed in the first place, and why it would be important to automate the process of software verification. We then briefly outline the contributions made in this thesis that allow us to inch closer to this ambitious goal.

1.1 The Need For Software Verification

Over the past decades, software has become ever more important in our daily lives — we use software when we grab our mobile or smart phone, when we gesture on our tablets, when we work on our computer in the office or at home, and even in the household when we use the washer, dryer, or modern kitchen utensils. Besides private life, software systems are nowadays also ubiquitous in practically every part of society, be it traffic control on streets, rails, water, or in the air, in the medical, financial, or public sector, or in the industry. While a software malfunction of one's private device is considered a mild annoyance, the failure of a software system running in a manufacturing plant, or errors in an air-traffic control system may have catastrophic consequences.

Testing is the predominant way to increase the confidence that a piece of software behaves correctly, and by combining several forms of testing [91], such as unit testing, integration testing, system testing, user acceptance testing, and others, one can reach a level of test coverage such that the software under test can be considered error free. But for a sufficiently complex program, no matter how many test cases exist, testing can in general only be used to reveal the presence of bugs, but never to prove their absence [51].

Also due to the inherent incompleteness of testing, fierce bugs may remain existent in software while it is already running on production systems, and until these bugs are finally fixed they remain to be a threat, either because the software simply does not work as intended, or because the bug can be exploited by an attacker to, e. g., compromise the whole system where the faulty software is deployed.

In the last couple of years, several extremely dangerous bugs where found in popular software. For example, the bug informally known as "Heartbleed" [1] allows

[1] `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160`

data theft and impersonation of services and users through a vulnerability in the widely used OpenSSL cryptographic software library. The security bug informally known as “Shellshock”[2] allows an attacker to execute arbitrary code on a machine running a vulnerable version of the Unix Bash shell, a program running on countless servers accessible through the internet. By exploiting the bug informally known as “Stagefright”[3] an attacker can gain elevated access and execute remote code on vulnerable versions of the Android operating system, to date the most popular operating system for smart phones. A bug known informally as “goto fail”[4] allows man-in-the-middle attacks against computers running the Mac OS X operating system. Finally, home and office computers are regularly threatened while connected to the internet, due to critical flaws in the popular browser plug-ins for Java[5] or Flash[6], which are actively used by attackers to gain unauthorized access to these machines.

These examples demonstrate that, especially for complex, safety-critical software, it is desirable to prove that this software is correct and free of errors. With software *verification* one is able to *prove* the correctness of a piece of software[7]. This can be achieved by formal verification [58, 70], a manual process similar to theorem proving, or by interactive systems [82, 93], which, in their core, make use of the fundamental principles from formal verification, such that the user and verification system assist each other to obtain proofs for complex software.

With software systems getting more and more complex, the interest for software verification is also on the rise, and ideas and tools for *automatic* software verification have become a central part of research in computer science, and with this thesis we try to contribute new ideas and tools for automatic software verification, as well.

1.2 Automatic Software Verification

In a large-scale, industrial setting, automatic software verification is the only realistic option [7, 27, 31, 35] to cope with the demand of software to be verified. In such a setting, a software verification tool, also referred to as software verifier, is tasked to answer the question if a given piece of software or program is correct, i. e., if it conforms to a given specification. Such a specification may state that for all possible executions of the verification task no division by zero may occur, or no assertion or exception may be raised, or no buffer may overflow. Typically, and also throughout this thesis, the specification is part of the verification task, and the specification is

[2]`https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271`
[3]`https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1538`
[4]`https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1266`
[5]`https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=java`
[6]`https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=flash`
[7]A counterexample is immediately available in case the software contains an error.

regarded as violated if there exists an execution of the verification task such that a specific statement in the verification task is reachable [10,14]. So in essence, checking the specification is reduced to checking the reachability of a program statement. This problem is also known as the *reachability problem*.

Unfortunately, in general the reachability problem is undecidable, as it can be reduced to the halting problem [104]. Due to this inherent complexity of software verification two distinct directions of research have developed. On the one hand there is software model checking, where for a verification task a model is computed, e. g., by enumerating all reachable states of this verification task, and then it is checked if any of the reachable states violates the specification. Classic software model checking [44], inspired by the success of hardware model checking, has the advantages that it is precise, in the sense that in theory it does not report false alarms, and that a found specification violation can be mapped easily to a concrete counterexample. However, for complex verification tasks, classic software model checking hardly does scale. So, on the other hand there is static analysis, which, not only throughout this thesis, is also referred to as program analysis or abstract interpretation [47]. This technique mitigates the complexity of software verification by computing a rather coarse over-approximation for a given verification task, which helps to reduce the computational complexity at the expense of being less precise, i. e., many false alarms may be reported.

The concept of configurable program analysis (CPA) [21] subsumes both these two approaches in one general framework, and it facilitates the fine-grained control over the verification process not only in this aspect. All contributions in this thesis are formalized towards coherence with the CPA framework, and the respective implementations are integrated in the open-source software verifier CPAchecker [8], and in the following we briefly outline these contributions.

1.3 Contributions

The contributions of this thesis are detailed chronologically in the respective chapters, i. e., a subsequent contribution represents an extension of the former contribution. Figure 1.1 depicts how the individual contributions build upon each other.

The purpose of the very first contribution stems from the idea to create an analysis that complements the predominant symbolic analyses, like for example verifiers that are based on predicate abstraction [9] [7,18,24].

[8] `http://cpachecker.sosy-lab.org/`

[9] Throughout this thesis we refer to an analysis from this class of analyses as *predicate analysis*.

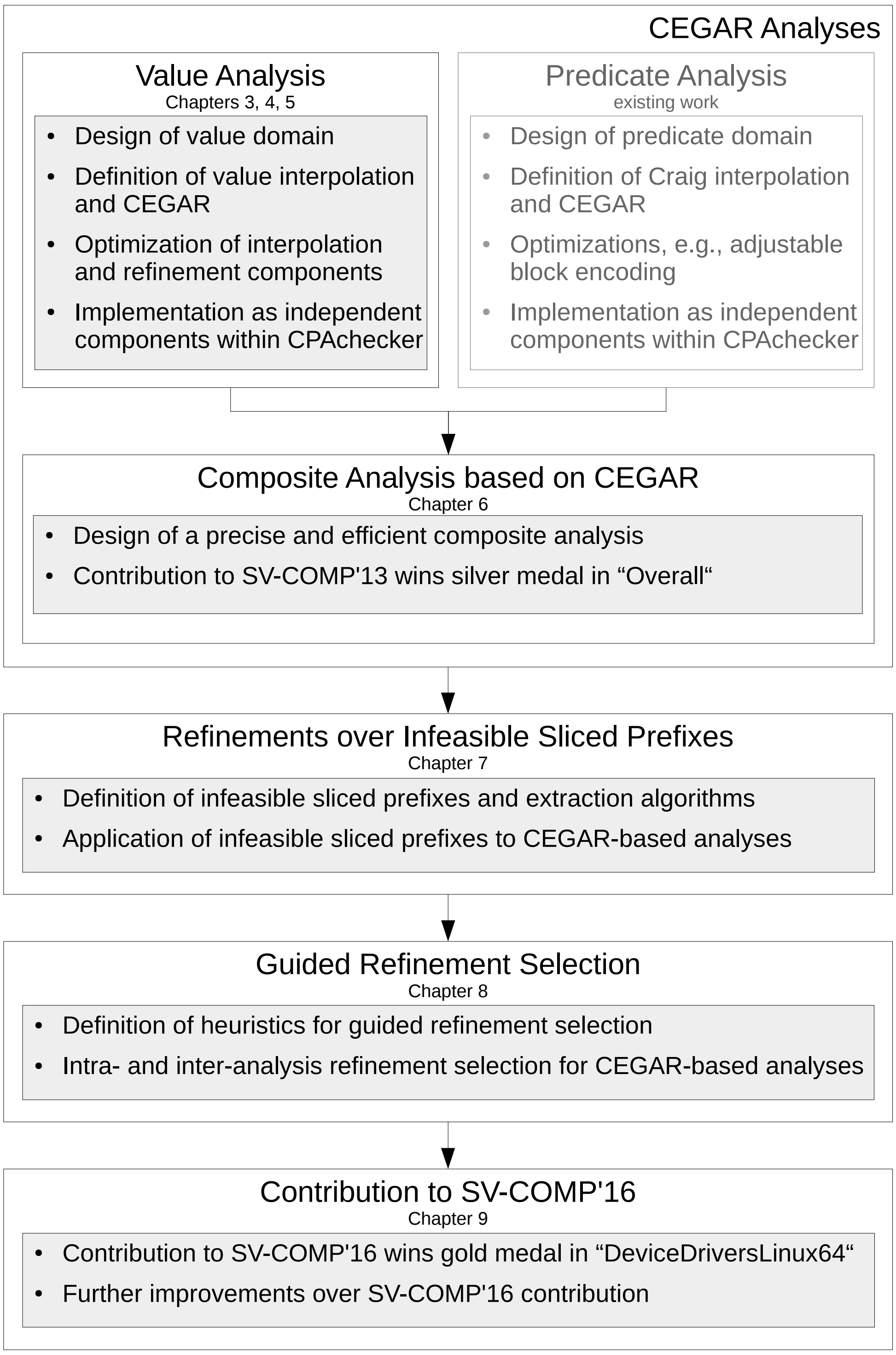

CEGAR Analyses
Value Analysis
Chapters 3, 4, 5
• Design of value domain
• Definition of value interpolation and CEGAR
• Optimization of interpolation and refinement components
• Implementation as independent components within CPAchecker
Predicate Analysis
existing work
• Design of predicate domain
• Definition of Craig interpolation and CEGAR
• Optimizations, e.g., adjustable block encoding
• Implementation as independent components within CPAchecker
Composite Analysis based on CEGAR
Chapter 6
• Design of a precise and efficient composite analysis
• Contribution to SV-COMP'13 wins silver medal in “Overall“
Refinements over Infeasible Sliced Prefixes
Chapter 7
• Definition of infeasible sliced prefixes and extraction algorithms
• Application of infeasible sliced prefixes to CEGAR-based analyses
Guided Refinement Selection
Chapter 8
• Definition of heuristics for guided refinement selection
• Intra- and inter-analysis refinement selection for CEGAR-based analyses
Contribution to SV-COMP'16
Chapter 9
• Contribution to SV-COMP'16 wins gold medal in “DeviceDriversLinux64“
• Further improvements over SV-COMP'16 contribution

Figure 1.1: Overview of the contributions made in this thesis

1.3.1 Value Analysis with CEGAR and Interpolation

Due to massive advancements in the fields of satisfiability theories (SAT) and satisfiability modulo theories (SMT) solving during the last two decades, symbolic analyses enjoy great success in the field of software verification [10, 11, 12, 13, 14]. Still, for some classes of verification tasks, the SAT and SMT solvers used as decision procedures in many verifiers are not always well suited or are simply overwhelmed by the complexity of some verification tasks.

Due to that, we designed an analysis that does not rely on external decision procedures, but instead borrows ideas from static analysis, concrete execution and explicit-state model checking. We called this analysis *value analysis*, because it tracks the concrete *values* of program variables in a domain we refer to as the *value domain*. The design and implementation of this analysis itself already proves to be beneficial, and we extended it by defining counterexample-guided abstraction refinement (CEGAR) and interpolation for the value domain. This did not only extend the successful application of the value analysis to more verification tasks, but the CEGAR and interpolation scheme of the value domain was also successfully applied to other domains similar to the value domain, e. g., to an octagon domain and a symbolic execution analysis. Hence, any advancements in the CEGAR technique are now applicable to predicate analyses, as well as to all analyses compatible with our CEGAR approach designed for the value domain.

Our paper covering this contribution was accepted for publication at FASE'13 [26].

1.3.2 Precise and Efficient Composite Analysis based on CEGAR

The central design decision of the value analysis is its simplicity, i. e., its reluctance to track any information symbolically. Therefore, especially for verification tasks that expose non-deterministic behavior, the value analysis alone is less precise than a symbolic analysis could be, and the value analysis may, by mistake, report specification violations for error-free verification tasks.

To counteract this imprecision, we conceived a novel *composite analysis*, featuring the value analysis and a predicate analysis, where both of the component analyses incorporate the CEGAR technique. In order to keep the composite analysis closer to the characteristics of the value analysis, we designed the refinement protocol of the composite analysis in such a way, that, by default, first the CEGAR and interpolation scheme of the value analysis is tasked to perform a refinement, and only if the expressiveness of the value domain is insufficient, then this gap is filled with the computational more expensive predicate analysis.

The result is an analysis that is both precise and efficient, as confirmed by the results of the International Competition on Software Verification (SV-COMP) where our verifier won the silver medal in three sub-categories as well as in the overall ranking [11] [10].

Our competition contribution was accepted for publication at SV-COMP'13 [83].

1.3.3 Refinements over Infeasible Sliced Prefixes

Despite the success of the composite analysis that features the value analysis and a predicate analysis, there remain verification tasks that neither of the analyses alone nor the composite approach solves. One reason for the verification process not converging in a timely fashion is because the facts extracted by the interpolation engine during CEGAR iterations are not always well suited for the analysis. For example, in case of the value analysis the interpolation engine might tell the analysis to track loop-counter variables, which often leads to state-space explosion and then to the divergence of the analysis. Similarly, for the predicate analysis the interpolation engine might tell the analysis again and again to track another inequality predicate about a loop-counter variable, but without finding a valid bound for this loop-counter variable, which eventually also leads to divergence of the analysis.

We designed an algorithm that extracts from one interpolation problem a set of interpolation problems, and thus giving the verifier more control over the interpolation process even in the case where external decision procedures like SMT solvers are used for interpolation. This novel contribution allows us to consider an interpolation problem as *optimization problem*, while also relaxing the black-box characteristics of interpolation engines.

Our paper covering this contribution was accepted for publication at FORTE'15 [30].

1.3.4 Guided Refinement Selection

With infeasible sliced prefixes being available an interpolation problem may be seen as optimization problem, but it is yet unclear how to capitalize from this insight.

Therefore, we extracted various (quality) criteria for refinements, which allow us to formulate heuristics to compute a ranking for a set of refinements and, based on the heuristics, select a refinement according to that resulting ranking. Based on a huge evaluation performed both for the value as well as for the predicate analysis, we investigated the effect of *guided refinement selection* using different heuristics. We furthermore devised the concept of *inter-analysis* refinement selection, where for a composite analysis we query heuristics to decide which of the component analyses is best to be refined in order to allow the composite analysis to converge faster.

Our paper covering this contribution was accepted for publication at SPIN'15 [29].

[10] `http://sv-comp.sosy-lab.org/2013/results/index.php`

1.3.5 Contribution to SV-COMP'16

We incorporated all contributions listed above into CPAchecker, and we participated with the respective configuration in SV-COMP'16 [14].

Our competition contribution was accepted for publication at SV-COMP'16 [84], and our verifier won the gold medal in the category `DeviceDriversLinux64`.

1.3.6 Availability of Implementations and Experimental Data

All our implementations are integrated into CPAchecker, and they are freely and publicly available. In order to allow reproducibility of all evaluations in this thesis, we provide the full results and raw data on our supplementary web page available at `http://www.sosy-lab.org/research/phd/loewe/`.

1.4 Structure of the Thesis

The rest of this thesis is organized as follows:

- After this brief introductory chapter some more background on (automatic) software verification is presented in Chapter 2.
- The value analysis is introduced in Chapter 3.
- The concepts of interpolation and CEGAR for the value domain are detailed in Chapter 4.
- In Chapter 5 we discuss optimizations for the newly designed interpolation procedure and the CEGAR technique of the value domain, and also comment on the endeavor of our further research on the topic of value domain refinement.
- Chapter 6 holds the details around the composite analysis, which features the value analysis and a predicate analysis.
- After that, the concept of infeasible sliced prefixes is introduced in Chapter 7, and in Chapter 8 this is extended to address intra- and inter-analysis refinement selection.
- In Chapter 9 we discuss our contribution to SV-COMP'16 and its current state after further development.
- Chapter 10 provides a summary of the thesis as well as a brief outlook on further research directions.

2 Background

The previous chapter emphasized the importance of software correctness, and we outlined several techniques that are available for improving the quality of software, such as testing, theorem proving, or automatic software verification, with the latter being in the focus of this thesis. In this chapter we now turn the attention to the theoretical background of software verification. This is needed to explain the ideas and concepts for automatic software verification presented in this thesis.

Hence, this chapter will introduce the basics around programs, the control-flow automaton (CFA) of programs, as well as define the semantics of programs, paths, and operations. This builds the basis for defining the CPA concept, which incorporates an abstract domain, a transfer relation, and other operators, that together allow formulating a reachability algorithm, namely the CPA algorithm. Finally, one possible instantiation of the CEGAR algorithm is presented, along with the notion of interpolation, so that when all is put together, flexible and efficient algorithms for automatic software verification can be designed.

2.1 Programs, Control-Flow Automaton, and Semantics

We restrict the presentation to a simple imperative programming language, where all operations are either assignments or assume operations, and all program variables range over integers.

A program is represented by a CFA A, with $A = (L, l_0, G)$, which consists of a set L of program locations, modeling the program counter, an initial program location $l_0 \in L$, representing the program entry, and a set $G \subseteq L \times Ops \times L$ of control-flow edges, modeling the operations that are executed when control flows from one program location to the next. The set of program variables that occur in operations from Ops is denoted by X.

A *verification task* $P = (A, l_e)$ consists of a CFA A representing the program, and a target location $l_e \in L$, which represents the specification, i. e., "the program must not reach location l_e".

A *concrete data state* of a program is a variable assignment $cd : X \to \mathbb{Z}$, which assigns to each program variable a value from the set $\mathbb{Z}$ of integer values.

A *concrete state* of a program is a pair (l, cd), where $l \in L$ is a program location and cd is a concrete data state. The set of all concrete states of a program is called C.

A *region* ρ represents a set concrete states for which the following holds: $\rho \subseteq C$.

We denote the *definition range* for a function f as $\mathsf{def}(f) = \{x \mid \exists y : f(x) = y\}$, and the *restriction* of a function f to a new definition range Y by $f_{|Y} = f \cap (Y \times \mathbb{Z})$.

Each edge $g \in G$ defines a labeled transition relation $\xrightarrow{g} \subseteq C \times \{g\} \times C$. The complete transition relation $\rightarrow$ is the union over all control-flow edges: $\rightarrow = \bigcup_{g \in G} \xrightarrow{g}$. We write $c \xrightarrow{g} c'$ if $(c, g, c') \in \rightarrow$, and $c \rightarrow c'$ if there exists an edge g with $c \xrightarrow{g} c'$.

A *path* σ is a sequence $\langle (op_1, l_1), \dots, (op_n, l_n) \rangle$ of pairs of an operation and a location. The path σ is called *program path*, if the path σ represents a syntactic walk through the CFA, i. e., for every i with $1 \leq i \leq n$ there exists an edge $g = (l_{i-1}, op_i, l_i)$, and l_0 is the initial program location. The result of appending the pair (op_n, l_n) to a path $\sigma = \langle (op_1, l_1), \dots, (op_m, l_m) \rangle$ is defined as $\sigma \wedge (op_n, l_n) = \langle (op_1, l_1), \dots, (op_m, l_m), (op_n, l_n) \rangle$, and the result of appending the sequence $\langle (op_n, l_n), \dots, (op_p, l_p) \rangle$ to a path $\sigma = \langle (op_1, l_1), \dots, (op_m, l_m) \rangle$ is defined as $\sigma \wedge \langle (op_n, l_n), \dots, (op_p, l_p) \rangle = \langle (op_1, l_1), \dots, (op_m, l_m), (op_n, l_n), \dots, (op_p, l_p) \rangle$.

A *constraint sequence* $\gamma_\sigma = \langle op_1, \dots, op_n \rangle$ is defined through the sequence of operations occurring in the path $\sigma = \langle (op_1, l_1), \dots, (op_n, l_n) \rangle$ it is associated with.

2.2 Configurable Program Analysis

All contributions in this thesis are based on the CPA concept [21] and its extensions to CPA+ [22]. In the following, if we refer to CPA, this always denotes CPA+. A CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ is an abstract reachability analysis that operates on a CFA and consists of an abstract domain D, a set of precisions Π, a transfer relation $\rightsquigarrow$, and the three operators merge, stop, and prec. The following paragraphs, describing each individual component of such a CPA, are taken from existing work [22].

2.2.1 Abstract Domain

The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ is defined by the set C of concrete states, the semi-lattice $\mathcal{E}$ of abstract states, and a concretization function $[\![\cdot]\!]$.

The semi-lattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ consists of the (possibly infinite) set E of abstract domain elements, the top element $\top \in E$, the bottom element $\bot \in E$, the partial order $\sqsubseteq \subseteq E \times E$, and the join operator $\sqcup : E \times E \rightarrow E$.

The function $\sqcup$ yields the least upper bound for two lattice elements, and the symbols $\top$ and $\bot$ denote the least upper bound and the greatest lower bound of the set E, respectively.

The concretization function $[\![\cdot]\!] : E \rightarrow 2^C$ assigns to each abstract state e its meaning, i. e., the set of concrete states that it represents.

For soundness of the program analysis, the abstract domain has to fulfill the following requirements:

(a) $[\![\top]\!] = C$ and $[\![\bot]\!] = \emptyset$

(b) $\forall e, e' \in E : e \sqsubseteq e' \Rightarrow [\![e]\!] \subseteq [\![e']\!]$

(c) $\forall e, e' \in E : [\![e \sqcup e']\!] \supseteq [\![e]\!] \cup [\![e']\!]$
(the join operator is precise or over-approximates $[\![e]\!] \cup [\![e']\!]$)

Requirement (c) is implied by (b) and the fact that $\sqcup$ yields the least upper bound.

2.2.2 Precision

The set of precisions Π determines the possible precisions $\pi \in \Pi$ of the abstract domain. The program analysis uses the elements π from Π to keep track of different precisions for different abstract states from E. As such, a pair $(e, \pi) \in E \times \Pi$ is called abstract state e with precision π. The operators of the abstract domain are parametric in the precision. In the remainder of this thesis the precision may be omitted if clear from context or when it is unnecessary for describing the characteristics of a component.

2.2.3 Transfer Relation

The transfer relation $\rightsquigarrow \subseteq E \times G \times E \times \Pi$ assigns to each abstract state e possible new abstract states e' with precision p which are abstract successors of e, and each transfer is labeled with a control-flow edge g. We write $e \overset{g}{\rightsquigarrow} (e', \pi)$ if $(e, g, e', \pi) \in \rightsquigarrow$, and $e \rightsquigarrow (e', \pi)$ if there exists an edge g with $e \overset{g}{\rightsquigarrow} (e', \pi)$.
The transfer relation has to fulfill the following requirement:

(d) $\forall e \in E, g \in G, \pi \in \Pi :$
$\bigcup_{e \overset{g}{\rightsquigarrow} (e', \pi)} [\![e']\!] \;\supseteq\; \bigcup_{c \in [\![e]\!]} \{c' \mid c \xrightarrow{g} c'\}$
(the transfer relation over-approximates operations for every fixed precision)

2.2.4 Merge Operator

The merge operator $\mathsf{merge} : E \times E \times \Pi \to E$ weakens the second parameter using information from the first parameter, and potentially returns a new abstract state with the precision given as third parameter.
The merge operator has to fulfill the following requirement:

(e) $\forall e, e' \in E, \pi \in \Pi : e' \sqsubseteq \mathsf{merge}(e, e', \pi)$
(the result of merge can only be more abstract than the second parameter)

There are two straight-forward implementations for merge. One implementation, to which we refer to as merge_{sep}, does not combine states at all, but instead just returns the second parameter unchanged, i. e., $\mathsf{merge}_{sep}(e, e', \pi) = e'$. The other approach, called merge_{join}, implements the merge operator based on the join operator $\sqcup$ of the lattice, i. e., $\mathsf{merge}_{join}(e, e', \pi) = e \sqcup e'$. However, note that, with respect to the lattice, the result of $\mathsf{merge}(e, e', \pi)$ may be anything between e' and $\top$.

2.2.5 Stop Operator

The termination check $\mathsf{stop} : E \times 2^E \times \Pi \to \mathbb{B}$ checks if the abstract state given as the first parameter with the precision given as the third parameter, is subsumed by the set of abstract states given as the second parameter. The termination check can, for example, go through the elements of the set R that is given as the second parameter and search for a single element that, in accordance to the partial order $\sqsubseteq$, subsumes the first parameter, or —if D is a power-set domain [1]— can join the elements of R to check if R subsumes the first parameter.
The termination check has to fulfill the following requirement:

(f) $\forall e \in E,\ R \subseteq E,\ \pi \in \Pi :$
$\mathsf{stop}(e, R, \pi) = \mathit{true} \implies [\![e]\!] \subseteq \bigcup_{e' \in R} [\![e']\!]$
(if an abstract state e is considered to be *covered* by R, then every concrete state represented by e is represented by some abstract state from R)

Hence, two straight-forward implementations of this operator are $\mathsf{stop}_{sep}(e, R) = (\exists e' \in R : e \sqsubseteq e')$ and $\mathsf{stop}_{join}(e, R) = (e \sqsubseteq \bigsqcup R)$.

2.2.6 Precision-Adjustment Operator

The *precision-adjustment operator* $\mathsf{prec} : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$ computes a new abstract state and a new precision for a given abstract state with precision and a set of abstract states with precision. The precision-adjustment operator is applied after the transfer relation $\rightsquigarrow$ and may perform widening of the abstract state, in addition to changing the precision.
The precision-adjustment operator has to fulfill the following requirement:

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

2.3 CPA Algorithm

Based on the components of the CPA concept from above, one can formulate the CPA algorithm (cf. Algorithm 1), which may serve as reachability algorithm.

[1]A *power-set domain* is an abstract domain such that $[\![e \sqcup e']\!] = [\![e]\!] \cup [\![e']\!]$.

Algorithm 1: $\mathsf{CPA}(\mathbb{D}, R_0, W_0)$ (taken from [26])

```
Input     : a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$,
            an initial set $R_0 \subseteq (E \times \Pi)$ of abstract states with their precision,
            a subset $W_0 \subseteq R_0$ of frontier abstract states with their precision
Output    : a pair with (1) a set of reachable abstract states with their precision
                and (2) a set of frontier abstract states with their precision
Variables : a set $reached \subseteq E \times \Pi$ and a set $waitlist \subseteq E \times \Pi$

1  $reached := R_0$
2  $waitlist := W_0$
3  while $waitlist \neq \emptyset$ do
4      choose $(e, \pi)$ from $waitlist$
5      $waitlist := waitlist \setminus \{(e, \pi)\}$
6      foreach $e'$ with $e \rightsquigarrow (e', \pi)$ do
7          $(\hat{e}, \hat{\pi}) = \mathsf{prec}(e', \pi, reached)$
8          if $\mathsf{isTargetState}(\hat{e})$ then
9              return $(reached \cup \{(\hat{e}, \hat{\pi})\}, waitlist \cup \{(\hat{e}, \hat{\pi})\})$
10         foreach $(e'', \pi'') \in reached$ do
11             $e_{new} := \mathsf{merge}(\hat{e}, e'', \hat{\pi})$
12             if $e_{new} \neq e''$ then
13                 $reached := (reached \cup \{(e_{new}, \hat{\pi})\}) \setminus \{(e'', \pi'')\}$
14                 $waitlist := (waitlist \cup \{(e_{new}, \hat{\pi})\}) \setminus \{(e'', \pi'')\}$
15         if $\neg\mathsf{stop}(\hat{e}, \{e | (e, \cdot) \in reached\}, \hat{\pi})$ then
16             $reached := reached \cup \{(\hat{e}, \hat{\pi})\}$
17             $waitlist := waitlist \cup \{(\hat{e}, \hat{\pi})\}$
18 return $(reached, \emptyset)$
```

The CPA algorithm gets as input a CPA $\mathbb{D}$ as well as two sets of abstract states with their precision, namely the sets R_0 and W_0. Normally, the set R_0 is initially empty, and the set W_0 contains only the initial abstract state. These two sets are then used to initialize the sets *reached* and *waitlist*.

The set *reached* stores all abstract states with their precision that are found to be reachable. The set *waitlist* stores all abstract states with their precision that are not yet processed, i. e., the frontier, and during the course of the algorithm, i. e., during the state-space exploration, these two sets are being updated continuously.

After the initialization phase of the algorithm, an abstract state with its precision is chosen and removed from the *waitlist*, and its successors are computed according to the transfer relation. For each of these successor states, the algorithm adjusts the precision of the individual successor using the precision-adjustment operator prec. If the successor is a *target state*, i. e., it corresponds to the target location l_e, then the algorithm terminates, returning the pair of sets *reached* and *waitlist* —possibly as input for a subsequent precision refinement— as shown below (cf. Algorithm 2). Otherwise, using the operator merge, the abstract successor state is combined with each existing abstract state from *reached*. If the operator merge results in a new abstract state with information added from the new successor (the old abstract state is subsumed) then the old abstract state with its precision is replaced by the new abstract state with its precision in the sets *reached* and *waitlist*. If after the merge step the resulting new abstract state with its precision is covered by the set *reached* —that is, stop returned *true*— then further exploration of this abstract state is stopped. Otherwise, the abstract state with its precision is added to the set *reached* and to the set *waitlist*. Finally, once the set *waitlist* is empty, the set *reached* is returned in a pair along with the empty *waitlist*.

For simplicity, the collections *reached* and *waitlist* are represented as simple sets here, but mind that the *waitlist* is implemented as priority queue, such that choosing from the *waitlist* can be configured from the outside. Furthermore, on top of the set *reached* there exists an implementation of a graph structure that we call abstract reachability graph (ARG) [18], and which is sketched in the following section.

2.4 Abstract Reachability Graph

The ARG represents the abstract model of a verification task, and it results from running a CPA on the CFA of the respective verification task, for example by using the CPA algorithm from above. So essentially, the ARG represents the unrolling of a CFA with a specific CPA. Hence, ARGs are different for different input CFAs, but also for two identical input CFAs the resulting ARGs usually differ if the analyses are performed using different or differently configured CPAs.

Besides representing a region of concrete data states, e. g., through an abstract variable assignment (cf. Section 3.3) or a first-order formula [24], each abstract state contained in the ARG usually wraps further auxiliary state information, such as the respective program location and the call stack at which this abstract state has been computed.

In addition, the ARG encapsulates two more pieces of information that are of importance. First, it has encoded the (transitive) predecessor-successor relation between the abstract states, and second, it knows which pairs of abstract states are in a coverage relation. The predecessor-successor relation is a minimal requirement for CEGAR, because CEGAR operates on counterexamples and we need to provide these counterexamples, i. e., the paths leading from the initial state to a state identified as potential target state, to the CEGAR algorithm, which we introduce in the following.

2.5 Counterexample-Guided Abstraction Refinement

CEGAR is a technique for automatic, iterative refinement of an abstract model [43], which is based on the following four building blocks:

1. a *state-space exploration algorithm*, computing the abstract model,
2. a *precision*, defining the current level of abstraction,
3. a *feasibility check*, deciding if an error path is feasible, and
4. a *refinement procedure*, enabling the creation of a more precise abstract model.

Algorithm 2 shows one, general instantiation of an algorithm following the CEGAR approach. It uses the algorithm CPA (cf. Algorithm 1) as *state-space exploration algorithm*, and an abstract domain that is formalized as a CPA with dynamic precision adjustment $\mathbb{D}$ (cf. Section 2.2). The CPA uses a set E of abstract states and a set Π of *precisions*. As described above, the main purpose of the CPA algorithm is to compute

Algorithm 2: CEGAR($\mathbb{D}, (e_0, \pi_0)$) (taken from [30])

```
Input    : a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$,
           an initial abstract state $e_0 \in E$ with a precision $\pi_0 \in \Pi$
Output   : the verification result true, or false (with counterexample)
Variables: a set $reached \subseteq E \times \Pi$, a set $waitlist \subseteq E \times \Pi$,
           an error path $\sigma = \langle (op_0, l_0), \dots, (op_n, l_n) \rangle$

1  $reached := \{(e_0, \pi_0)\}$
2  $waitlist := \{(e_0, \pi_0)\}$
3  $\pi := \pi_0$
4  while $true$ do
5      $(reached, waitlist) := \mathsf{CPA}(\mathbb{D}, reached, waitlist)$
6      if $waitlist = \emptyset$ then
           // no error path found:  verdict is true
7          return true
8      $\sigma := \mathsf{ExtractErrorPath}(reached)$
9      if $\mathsf{IsFeasible}(\sigma)$ then
           // error path is feasible:  verdict is false, report bug
10         return false
11     else
           // error path is infeasible:  restart with refined precision
12         $\pi := \pi \cup \mathsf{Refine}(\sigma)$
13         $reached := \{(e_0, \pi)\}$
14         $waitlist := \{(e_0, \pi)\}$
```

the set *reached* and *waitlist*, which represent the current reachable abstract states with precisions and the frontier, respectively. In accordance to the CEGAR concept, the CPA algorithm is initially run with π_0 as coarse initial precision (usually $\pi_0 = \emptyset$). If all program states have been exhaustively checked, indicated by an empty *waitlist*, and no target state was reached, then the CEGAR algorithm terminates and reports the verdict `true`, i. e., the verification task is considered safe. If the CPA algorithm finds an error in the abstract state space, i. e., a counterexample for the given specification, then it stops and returns the yet incomplete sets *reached* and *waitlist*. Now the corresponding abstract error path σ is extracted from the set *reached* using the procedure ExtractErrorPath[2], and passed to the procedure IsFeasible for the *feasibility check*, where this extracted abstract error path is reevaluated with full precision. If the abstract error path σ is found to be feasible also with full precision, this means there exists a corresponding concrete error path, and this concrete error path represents a violation of the specification. Hence, the algorithm terminates, returning the verdict `false` and reporting the bug. If, however, the error path σ is found to be infeasible, i. e., it is not corresponding to a concrete program path, then the current precision π was too coarse and needs to be refined. The *refinement procedure* is represented here by the procedure Refine returning a precision that is strong enough to exclude this infeasible error path from future state-space explorations. This returned precision is used to extend the current precision π of the CPA algorithm, which re-computes the sets *reached* and *waitlist* based on this new, refined precision in the next iteration of the CEGAR loop.

The above describes the standard CEGAR approach, i. e., after each and every refinement the sets *reached* and *waitlist* are reset and the state-space exploration starts anew, then with the refined precision as returned by the procedure Refine. If combining CEGAR with the concept of lazy abstraction [69], the overall analysis may become more efficient. This is made possible by one of the key ingredients of lazy abstraction, namely that after a refinement only those parts of the sets *reached* and *waitlist* are removed that needed to be refined, while leaving the remainder of the explored state space unchanged. Hence, the state-space exploration continues from there using the newly refined precision, without the need of recomputing those parts of the state space that remain unaffected from this refinement anyway. For selectively removing parts of the sets *reached* and *waitlist* there must be means to obtain all descendants of an abstract state, as well as to restore the consistency of the coverage relation after a lazy refinement, i. e., CEGAR with lazy abstraction requires an ARG to be available. Our implementation is based on the CPAchecker which already provides these functionalities through the well-established ARGCPA (cf. Section 2.7).

[2]In the implementation the path is extracted using the predecessor-successor relation of the ARG.

2.6 Interpolation

As shown in the previous section, the CEGAR algorithm needs some implementation for the procedure Refine, which is responsible for refining the precision employed in the state-space exploration algorithm, such that the infeasible counterexamples get excluded one after the other. There exist several different possibilities for realizing such a refinement procedure, e. g., mining predicates [34], statically computing invariants [76], or invariant generation [54], but arguably, the most prominent and successful ones in the field of automatic software verification are based on (Craig) interpolation [10, 14, 49, 68, 87].

Craig interpolation is a technique from logics that yields for two contradicting formulas an interpolant formula that contains less information than the first formula, but is still expressive enough to contradict the second formula.

Formally, for a pair of formulas φ^- and φ^+ such that $\varphi^- \wedge \varphi^+$ is unsatisfiable, a Craig interpolant ψ is a formula that fulfills the following three requirements [49]:

1. the implication $\varphi^- \Rightarrow \psi$ holds,
2. the conjunction $\psi \wedge \varphi^+$ is unsatisfiable, and
3. ψ only contains symbols that occur in both φ^- and φ^+.

Such a Craig interpolant is guaranteed to exist for many useful theories, e. g., the theory of linear arithmetic, as implemented in several SMT solvers. Hence, for software verification, interpolation was first used in the domain of predicate abstraction [68], however, a later chapter will introduce the concept of interpolation also for the value domain [26].

Independent of the analysis domain, interpolants for infeasible error paths are useful for refining the precision of the state-space exploration algorithm, such that these infeasible error paths are excluded in subsequent CEGAR iterations.

2.7 CPAchecker as Verification Framework

CPAchecker [23] is an open-source project made available under the Apache 2.0 license. It is being developed by the members of the Software Systems Lab, led by Dirk Beyer, at the University of Passau. Both source code and binary releases are available from `http://cpachecker.sosy-lab.org`. CPAchecker is used by practitioners and researchers, for example at the Russian Academy of Science, at the universities of Darmstadt, Hamburg, Paderborn and Vienna, as well as at Verimag in Grenoble.

CPAchecker is designed as an extensible framework for software verification. It is written in Java and the main focus is laid on verification of sequential C programs. The framework uses the C parser from the Eclipse CDT project and offers interfaces to

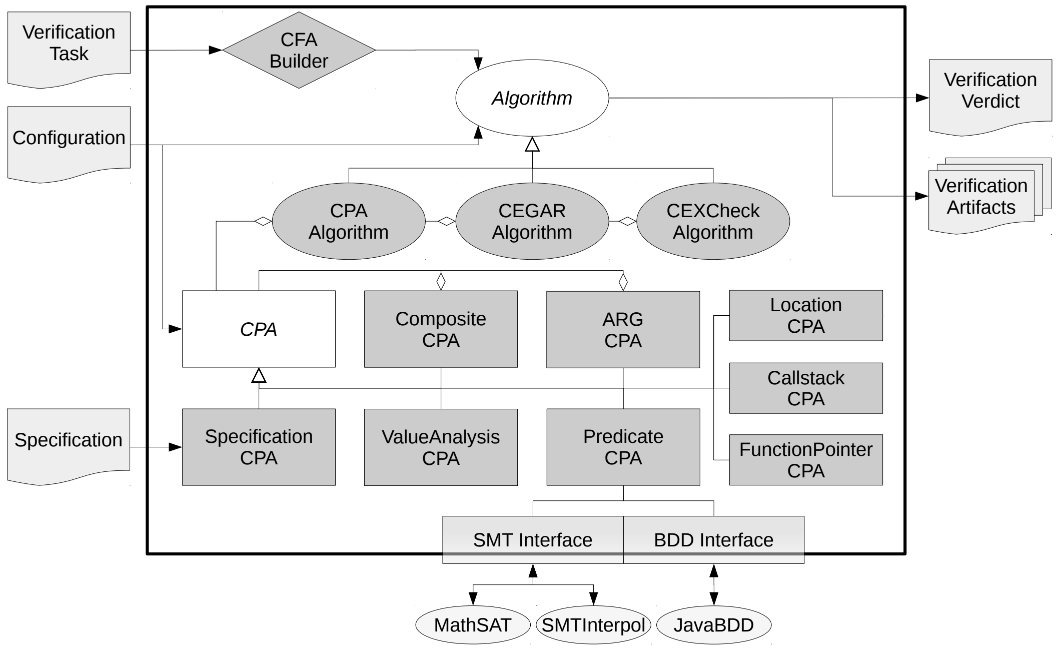


Figure 2.1: Overview of the architecture of the CPAchecker framework

several SMT solvers, e. g., MathSAT5 [41], SMTInterpol [40], and others, for solving and interpolating over formulae. The paramount design decision of CPAchecker is separation of concerns, thus, many of the standard tasks which are required for virtually any verification approach are implemented as individual, independent CPAs [21] within CPAchecker (cf. Figure 2.1). For example, the program counter is tracked by the LocationCPA, the call stack is modeled by the CallstackCPA, function pointers are resolved by the FunctionPointerCPA, and adherence to the specification is monitored by the SpecificationCPA. The principle of separating concerns is also kept for the main analyses, e. g., for the value analysis or for the predicate analysis, which represent the two main analyses described throughout this thesis. Both these analyses are also implemented as strictly decoupled CPAs within CPAchecker, namely as ValueAnalysisCPA and PredicateCPA, respectively.

Given a user-supplied configuration, any of the CPAs available within CPAchecker may be enabled on a per-demand basis and flexibly recombined to form a parallel composition of analyses by means of the CompositeCPA, which itself may be used to form an ARG-based analysis by wrapping it in the ARGCPA.

Any such CPA can then plugged into a configurable instance of an Algorithm, e. g., the CPAAlgorithm, to perform the state-space exploration for some verification task. Other algorithms, like the CEGARAlgorithm or the CEXCheckAlgorithm can be asked to wrap the CPAAlgorithm, simply by passing in the respective configuration to the CPAchecker framework.

CPAchecker has become one of the most successful tools for automatic software verification, winning a series of gold, silver and bronze medals in every edition of the SV-COMP, and the CPAchecker team was awarded the Kurt Gödel medal for their achievements in the field of automatic software verification.

Note that all contributions described in this thesis are integrated in the CPAchecker framework. The respective approaches have been thoroughly implemented, improved, optimized, and maintained over several years, adding to the overall value and success of the CPAchecker project.

3 Value Analysis

After having laid out the concepts of CPA and the CPA algorithm, we now define an analysis to be used as component analysis within the CPA framework. We call the analysis *value analysis*, as it tracks the concrete *values* for the program variables of a verification task.

3.1 Motivation

Current research lays a strong focus on concepts and ideas that propose to perform program analysis *symbolically* [103], and thus these analyses have to rely on some form of external decision procedure like a SAT or SMT solver. While the respective implementations of this approach perform well on a wide range of verification tasks [10, 11, 12, 13, 14], there are also many cases where a fully symbolic approach does not scale because the SAT or SMT solver is overwhelmed. Instead, tracking state information concretely makes the use of external decision procedures unnecessary, which leads to more efficient successor computations. This becomes magnified in presence of bit-operations, because the SMT solvers either must employ bit-precise reasoning, which comes at an even higher computational cost, or they have to over-approximate such operations which introduces imprecision to the analysis. In contrast, an analysis that tracks the values of program variables concretely may decide, and more importantly has the capability, to model bit-level or floating-point operations accurately.

Therefore, to contrast the predominant symbolic analyses, we conceive a *value analysis* that tracks only *concrete values* of program variables. However, when compared to symbolic approaches, this design decision does not only come with the advantages outlined above, but there are also a few important disadvantages to bear in mind. The simple state representation of the value analysis —it only tracks concrete assignments of program variables— leads to an analysis that is less precise in the presence of non-deterministic behavior, e. g., introduced due to program variables not being initialized or program variables being assigned from external function calls. Whereas symbolic approaches still can learn some state information in these cases, the value analysis is in general not able to derive any concrete value for a program variable in such a scenario. Moreover, enumerating all possible concrete values for all program variables can easily lead to a huge amount of abstract states, and the

```
 1 #include <assert.h>
 2 int main() {
 3   int a = 0;
 4   int b = 1;
 5   int c;
 6   b = a + b;
 7
 8   if (c) {
 9     a = 1;
10   }
11   else {
12     a = 2;
13   }
14
15   int f = a - b;
16   if (f < 0) {
17     assert(0);
18   }
19 }
```

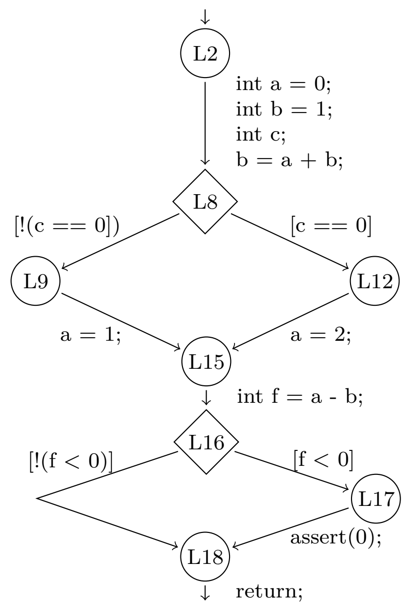


Figure 3.1: Simple verification task, with its corresponding CFA

problem of state-space explosion might render the analysis infeasible. Symbolic approaches often allow a more compact state-space representation, which then may help to avoid the problem of state-space explosion. All in all, it is interesting to see if the value analysis we envision is able to compete with symbolic analyses.

Before introducing the formalism for the analysis, we illustrate, on the small example shown in Figure 3.1, how the value analysis operates.

There, the program variables `int a` and `int b` are properly initialized, while the program variable `int c` is left uninitialized. Thus, in the assume operation `if (c)` in line 8, both branches, i. e., the two assignments `a = 1` and `a = 2` need to be taken into account by the analysis, so that the statement `int f = a − b` can be evaluated for each case. Only this way a complete judgment over the reachability of the `assert` statement is possible, and the verdict `true` of the verification task can be reported.

The final ARG, as it looks after having ran the value analysis, is visualized in Figure 3.2. Before the initial state, all program variables are mapped to $\top$, indicating that no valuations for any program variable are known. In the first basic block, the program variables `int a` and `int b` are initialized and set to 0 and 1, respectively. Because the program variable `int c` is left uninitialized, both branches from the assume operation in line 8 are explored. From the condition `[c == 0]` the analysis can assume the value 0 for the program variable `int c` in the `else` branch and

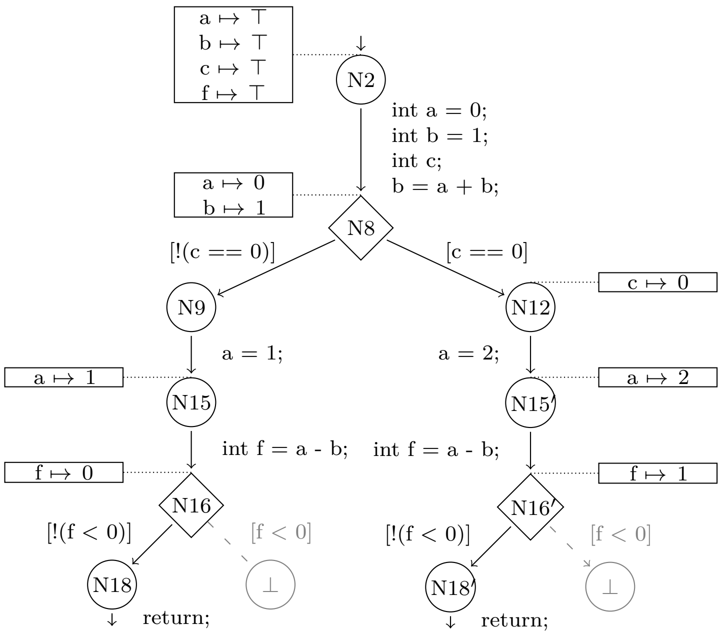


Figure 3.2: The ARG of the verification task from Figure 3.1, annotated with the assignments of program variables like the value analysis would compute them for the verification task

thereafter. Note that from the condition [c != 0] the analysis cannot derive any information, as it does not track inequalities. But despite this minor imprecision, in both branches after node N15 the valuation of the program variable **int** f can be evaluated to a deterministic value, which in both branches is not lower than 0, but 0 and 1, respectively. Consequently, the **else** branch of node N16 does not have a successor in neither of the two branches —indicated by the dashed line and faded color— which proves that the failing **assert** statement is unreachable and that the verdict for this verification task is `true`.

Now, after having introduced the operating mode of the value analysis, the next section discusses related work in that field. After that, we present the formal definitions, in order to pinpoint the semantics of the analysis precisely. These are then used to define the value analysis as individual component CPA in the CPA framework. This is followed by an evaluation of our value analysis, and the conclusion of this first chapter then details the strong points and weak spots of our first approach towards creating an efficient analysis.

3.2 Related Work

There are many different approaches to perform program analysis, and there is also a huge body of techniques and tools for program analysis, making a complete comparison with all available work impossible. Thus, we discuss here only the most prominent approaches for program analysis. In later chapters we compare our verifier to those verifiers that are made visible by their respective authors as successful participants in the SV-COMP editions of the last years [10,11,12,13,14].

The predominant analyses in the field of automatic software verification are based on predicate abstraction [7,9,60,86,100], or they perform bounded model checking [55,64,79,90] or symbolic execution [37,75,78,101]. Some approaches additionally use binary decision diagrams [6,32,36], or combine several of the mentioned techniques [4,85] in parallel or in sequence. Undoubtedly, these analyses have great success [10,11,12,13,14], and this success story is also due to the massive improvements in the field of automatic SAT and SMT solving [39,80,98] achieved during the last decade. All these approaches perform symbolic program analysis, and are therefore not that closely related to our approach, which is based on tracking concrete values of program variables. For a more detailed discussion on these approaches we refer the interested reader to the literature referenced above.

On the other side of the spectrum, there is the field of explicit-state software model checking. In its simplest form explicit-state software model checking enumerates all reachable states of a verification task, a characteristic that makes this approach seem closely related to value analysis that we conceive here. However, in the presence of non-determinism, e. g., stemming from uninitialized program variables or program variables assigned from external function calls, our approach always over-approximates the valuation of such a program variable. In contrast, explicit-state model checking may go as far as generating one state for each possible valuation of a program variable, i. e., for a non-deterministic program variable `int a` this alone would lead to 2^{32} states. The most renowned explicit-state model checkers are SPIN [71] and JAVA PATH FINDER [66]. JAVA PATH FINDER only supports the verification of Java programs, and SPIN was originally designed to verify properties of Promela [92] models only, and it is not possible to directly verify C source code with SPIN. But, as SPIN translates the Promela model internally to a C program, later version allow to embed fragments of C within Promela models. Additionally, when supplying a *manually* designed test harness description to the tool MODEX [73], then the later creates a Promela model from that, which then can be verified with SPIN. The more recent explicit-state model checker DIVINE [8,102] supports the verification of both C and C++ by integrating the highly universal LLVM compiler infrastructure project [81]. However, non-deterministic behavior is not restricted by DIVINE, i. e., DIVINE is explicit in non-determinism, so it actually does enumerate all 2^{32} possibilities of a non-deterministic program variable of type `int`. Besides that, explicit-state model checkers such as Spin, Java Path Finder, or Divine are primarily designed for concurrent systems, while our value analysis mainly targets reachability properties of single-threaded programs.

The main problem of program analyses based on either explicit or symbolic model checking is that of state-space explosion. Program analyses following the abstract interpretation framework [47] mitigate this problem to some extent, because whenever two branches in a program meet again, the defined join operator merges the two individual states available at the end of each branch, such that the result of the join subsumes the information from both branches. So, instead of the two individual branches only one has to be explored further, which lowers the number of abstract states to be enumerated, but at the cost of a lower precision which may lead to lots of false positives, i. e., incorrect `false` verdicts [21]. Note that we envision our value analysis to be path-sensitive, otherwise it would be too imprecise and almost identical to constant propagation [1]. Two well-known tools that are based on abstract interpretation are the static analyzer Astree [35] and the value analysis of the Frama-C platform [38].

This discussion of techniques and approaches is by no means intended to be complete, as there exist many more approaches for program analysis. For example, there is a plethora of work on pointer analyses, on analyses to check memory safety or verify concurrent programs, as well as analyses to reason about program termination. All these approaches are orthogonal to our value analysis, which is thought to run in a model-checking configuration and to reason over reachability properties in sequential C programs. In the following we will present the basic definitions and concepts of our value analysis and then embed it as CPA into the CPA framework.

3.3 Definitions

An *abstract data state*, formally defined as abstract variable assignment, represents a region of concrete data states.

An *abstract variable assignment* is either (1) a partial function $v : X \nrightarrow \mathbb{Z}$ mapping variables in its definition range to integer values, or (2) $\bot$, which represents no variable assignment (i. e., no value is possible, similar to the predicate *false* in logic). The special abstract variable assignment $\top = \emptyset$ does not map any variable to a value and is used as initial abstract variable assignment in a program analysis. We model abstract variable assignments as partial functions because specific variables, despite being declared in the program, may not be represented in abstract variable assignments. This can be due to the analysis not being able to determine a value for a variable, e. g., because a variable is left uninitialized or is assigned to the return value of an external function call, or because the analysis omitted a variable on purpose for reasons of abstraction.

The *definition range* of a partial function f is defined as $\mathsf{def}(f) = \{x \mid \exists y : f(x) = y\}$ and the *restriction* of a partial function f to a new definition range Y is defined as $f_{|Y} = f \cap (Y \times \mathbb{Z})$. For two partial functions f and f', $f(x) = y$ represents the predicate $(x, y) \in f$, and $f(x) = f'(x)$ represents the predicate $\exists c : (f(x) = c) \wedge (f'(x) = c)$.

An abstract variable assignment v represents the set $[\![v]\!]$ of all concrete data states cd for which v is valid, or formally, (1) for $v = \bot$, $[\![v]\!] = [\![\bot]\!] = \emptyset$, and (2) for all $v \neq \bot$, $[\![v]\!] = \{cd \mid \forall x \in \mathsf{def}(v) : v(x) = cd(x)\}$.

The abstract variable assignment $\bot$ is called *contradicting.* The *implication* for abstract variable assignments is defined as follows: v implies v' (written $v \implies v'$) if (1) $v = \bot$, or (2) for all variables $x \in \mathsf{def}(v')$ we have $v(x) = v'(x)$. The *conjunction* for abstract variable assignments v and v' is defined as:

$$v \wedge v' = \begin{cases} \bot & \text{if } v = \bot \text{ or } v' = \bot \text{ or } (\exists x \in \mathsf{def}(v) \cap \mathsf{def}(v') : \neg(v(x) = v'(x))) \\ v \cup v' & \text{otherwise} \end{cases}$$

An *abstract state* of a program is a pair (l, v), representing the following set of concrete states: $\{(l, cd) \mid cd \in [\![v]\!]\}$.

The *semantics of an operation $op \in Ops$* is defined by the strongest post-operator $\widehat{\mathsf{SP}}_{op}(\cdot)$. Given an abstract variable assignment v, the abstract variable assignment $v' = \widehat{\mathsf{SP}}_{op}(v)$ is the resulting abstract variable assignment when executing op on the input v. Formally, given an abstract variable assignment v and an assignment operation $x := exp$, we have (1) $\widehat{\mathsf{SP}}_{x:=exp}(v) = \bot$ if $v = \bot$, or (2) $\widehat{\mathsf{SP}}_{x:=exp}(v) = v_{|X \setminus \{x\}} \wedge v_x$ with

$$v_x = \begin{cases} \{(x, c)\} & \text{if } c \in \mathbb{Z} \text{ is the result of the arithmetic evaluation of } exp_{/v} \\ \emptyset & \text{otherwise (if } exp_{/v} \text{ cannot be evaluated)} \end{cases}$$

where $exp_{/v}$ denotes the interpretation of the expression exp for the abstract variable assignment v. Given an abstract variable assignment v and an assume operation $[p]$, we have (1) $\widehat{\mathsf{SP}}_{[p]}(v) = \bot$ if $v = \bot$ or the predicate $p_{/v}$ is unsatisfiable, or (2) we have $\widehat{\mathsf{SP}}_{[p]}(v) = v \wedge v_p$, with $v_p = \Big\{(x, c) \in (X \setminus \mathsf{def}(v) \times \mathbb{Z}) \;\Big|\; p_{/v} \implies (x = c)\Big\}$ and $p_{/v} = p \wedge \bigwedge_{y \in \mathsf{def}(v)} y = v(y)$, i. e., c is the only possible assignment for the variable x so that the formula p is satisfied.

The *semantics of a path* $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ is defined as the successive application of the strongest post-operator to each operation of the corresponding constraint sequence γ_σ, formally, $\widehat{\mathsf{SP}}_{\gamma_\sigma}(v) = \widehat{\mathsf{SP}}_{op_n}(\ldots \widehat{\mathsf{SP}}_{op_1}(v) \ldots)$. The abstract state that results from running a program path σ is represented by the pair $(l_n, \widehat{\mathsf{SP}}_{\gamma_\sigma}(\top))$, where $\top$ is the initial abstract variable assignment that is available at the initial program location. A path σ is called *feasible* if $\widehat{\mathsf{SP}}_{\gamma_\sigma}(\top)$ is not contradicting, i. e., $\widehat{\mathsf{SP}}_{\gamma_\sigma}(\top) \neq \bot$.

A concrete state (l_n, cd_n) is *reachable*, if there exists a feasible program path $\sigma = \langle(op_1, l_1), \dots, (op_n, l_n)\rangle$, with $cd_n \in [\![\widehat{\mathsf{SP}}_{\gamma_\sigma}(\top)]\!]$, and a location $l \in L$ is reachable if there exists a concrete data state cd such that (l, cd) is reachable.

A program, or verification task, is considered to have the verdict `true`, if l_e is not reachable. In this case the specification of the program is considered to be satisfied. A program path $\sigma = \langle(op_1, l_1), \dots, (op_n, l_e)\rangle$, for which $\widehat{\mathsf{SP}}_{\gamma_\sigma}(\top) \neq \bot$ is called *error path*.

3.4 Value Analysis as CPA

In the following, we define a component CPA for tracking concrete integer values of program variables. In order to obtain a meaningful analysis we need to construct a composite CPA, which consists of this component CPA for tracking concrete values of program variables, and a few more component CPAs, e. g., a CPA for tracking the program counter, as described in the literature [22]. For the composite CPA, the general definitions of the abstract domain, the transfer relation, and the other operators are also available from the literature [22]. The composition is automatically taken care of by the CPA framework.

The CPA for the value analysis is defined as $\mathbb{C} = (D_\mathbb{C}, \Pi_\mathbb{C}, \rightsquigarrow_\mathbb{C}, \mathsf{merge}_\mathbb{C}, \mathsf{stop}_\mathbb{C}, \mathsf{prec}_\mathbb{C})$ and consists of the following components [26].

3.4.1 Abstract Domain

The abstract domain $D_\mathbb{C} = (C, \mathcal{V}, [\![\cdot]\!])$ contains the set C of concrete data states, and uses the semi-lattice $\mathcal{V} = (V, \top, \sqsubseteq, \sqcup)$, which consists of the set $V = (X \rightarrowtriangle \mathbb{Z}) \cup \{\top, \bot\}$ of abstract variable assignments, where V, with $\mathbb{Z}$ denoting the set of integer values, induces the flat lattice over the integer values. The top element $\top \in V$ is the abstract variable assignment that holds no specific value for any program variable, and the bottom element $\bot \in V$ is the abstract variable assignment which models that there is no value assignment possible, i. e., a state that cannot be reached in an execution of the program. The partial order $\sqsubseteq \subseteq V \times V$ is defined as $v \sqsubseteq v'$, if (1) $v = \bot$, or (2) $v' = \top$, or (3) $\mathsf{def}(v') \subseteq \mathsf{def}(v)$ and $v(x) = v'(x)$ holds for all $x \in \mathsf{def}(v')$. The join $\sqcup : V \times V \to V$ yields the least upper bound for two abstract variable assignments. The concretization function $[\![\cdot]\!] : V \to 2^C$ assigns to each abstract variable assignment v its meaning, i. e., the set of concrete data states that it represents.

3.4.2 Precision

The precision of the analysis controls the level of abstraction employed during the analysis. The set of precisions $\Pi_\mathbb{C}$ consists of precisions π, where each precision π, with $\pi : L \mapsto 2^X$, specifies at which program locations $l \in L$ which program variables $x \in X$ are being tracked. For example, the precision $\pi(l) = X$ for all $l \in L$, also called the full precision, signals the analysis to track all program variables at all program locations, and a precision $\pi(l) = \emptyset$ for all $l \in L$, also called the empty precision, signals the analysis to track no program variables at all. Note, that we sometimes refer to the precision without explicitly giving the parameter $l \in L$. In such a case we assume $\pi(l) = \pi(l') = \xi \subseteq X$ for all $l, l' \in L$.

3.4.3 Transfer Relation

The transfer relation $\rightsquigarrow_{\mathbb{C}}$ has the transfer $v \overset{g}{\rightsquigarrow} (\widehat{\mathsf{SP}}_{op}(v), \pi)$, with $g = (\cdot, op, \cdot)$.

3.4.4 Merge Operator

The merge operator does not combine elements when control flow meets, i. e., $\mathsf{merge}_{\mathbb{C}}(v, v', \pi) = v'$.

3.4.5 Stop Operator

The stop operator considers abstract states individually, i. e., $\mathsf{stop}_{\mathbb{C}}(v, R, \pi) = (\exists v' \in R : v \sqsubseteq v')$.

3.4.6 Precision-Adjustment Operator

The precision-adjustment operator computes a new abstract state with precision based on the abstract state v and the precision π by restricting the abstract variable assignment v to those program variables that appear in π, formally, $\mathsf{prec}(v, \pi, R) = (v_{|\pi}, \pi)$. Mind that, in this analysis instance, there is not yet a procedure available to refine the precision π. Therefore, in practice, one is bound to use the full precision $\pi = X$ as (initial) precision, and therefore prec can be simplified to $\mathsf{prec}(v, \pi, R) = \mathsf{prec}(v, X, R) = (v_{|X}, \pi) = v$, i. e., there is no actual abstraction computation performed, because prec returns the original abstract variable assignment unchanged.

Before the next chapter introduces the required concepts to allow a meaningful precision-adjustment operator, the next section first focuses on the evaluation of the value analysis operating on a full precision.

3.5 Evaluation

Based on the definitions in this chapter, we implemented the ValueAnalysisCPA as a component CPA in the verification framework CPAchecker. Together with a few more component CPA s (e. g., a LocationCPA [21, 22] for tracking the program counter) we can formulate a CompositeCPA [21, 22] in order to obtain a complete program analysis for tracking concrete values of program variables. In this section we present a thorough evaluation of this analysis.

3.5.1 Setup

For benchmarking we use machines equipped with two Intel Xeon E5-2650v2 eight-core CPUs with 2.6 GHz and 135 GB of main memory, running Ubuntu 14.04 as operating system. In accordance to the rules of SV-COMP [10, 14], we limit each run to 900 s of CPU time, 15.0 GB of memory, and we execute each verification run on four CPU cores. As benchmarking framework we rely on BenchExec [1] for ensuring precise and reproducible results [28], such that the experimental setup is identical to the evaluation environment [2] advocated by the jury and the organizers of SV-COMP'16 [14].

3.5.2 Benchmarks

In order to allow a thorough evaluation of our approach, we use verification tasks from the official repository of the SV-COMP benchmark suite [3], as used in the latest edition of SV-COMP [14]. From the 6 661 available tasks, we only select verification tasks that deal with reachability properties, while disregarding verification tasks for memory safety, overflows or termination properties. We further omit verification tasks specifically designed for concurrency, dynamic data structures and pointers, or recursion, because the implementation of the value analysis does not have complete support for these features. This leaves us with 4 283 verification tasks to experiment with. In Table 3.1 we present an overview of the SV-COMP categories and the respective number of verification tasks that are included in our evaluation [4].

Category	Total	Evaluated
BitVectorsReach	48	48
ControlFlow	48	48
DeviceDriversLinux64	2 120	2 120
ECA	1 140	1 140
Floats	81	81
Loops	141	141
ProductLines	597	597
Sequentialized	261	62
Simple	46	46
ArraysMemSafety	65	0
ArraysReach	118	0
BitVectorsOverflows	12	0
Concurrency	1 016	0
HeapMemSafety	158	0
HeapReach	81	0
Recursive	98	0
Termination	631	0
Overall	6 661	4 283

Table 3.1: Overview of the verification tasks from SVCOMP'16 as used in this evaluation

[1] `https://github.com/dbeyer/benchexec`
[2] `http://sv-comp.sosy-lab.org/2016/systems.php`
[3] `https://github.com/sosy-lab/sv-benchmarks/releases/tag/svcomp16`
[4] For descriptions of verification tasks see `http://sv-comp.sosy-lab.org/2016/benchmarks.php`

3.5.3 Configuration

For evaluating the value analysis we use CPAchecker in trunk revision 20 406 —from March the 21st, 2016— and we start CPAchecker with the configuration named `valueAnalysis-Plain`. This configuration tells CPAchecker to run the standard value analysis without descending in recursive function calls and without performing counterexample checks using a secondary analysis [19, 85]. While the latter feature would help discarding incorrect `false` verdicts reported by the value analysis, so far, we are not interested in improving the precision of the analysis but rather want to evaluate how efficient and precise it is for itself.

In order to allow reproducibility of the evaluation, an example for a complete command line to run the value analysis as well as the full results and raw data are available on our supplementary web page[5].

3.5.4 Results

We now present the results of running this configuration of the value analysis on the given benchmarks in the environment as described above.

Verification Effectiveness of the Value Analysis

We discuss in detail only a few interesting aspects regarding the verification effectiveness of the value analysis. For a complete overview of the characteristics regarding the verification effectiveness of the value analysis on our benchmark set, we point the reader to Table 3.2.

Solved Verification Tasks In total, the value analysis solves 2 658 out of the 4 283 verification tasks. This means, that for almost two-thirds of the verification task —for 62 % to be exact— the value analysis obtains a result of either `true` or `false`. The set of all solved tasks contains all tasks for which a correct `true`, correct `false`, incorrect `true`, or an incorrect `false` verdict was obtained.

Incorrect True Verdicts The value analysis does not return a single incorrect `true` verdict for the whole benchmark set, i. e., for the verification tasks we considered in the experiment, the value analysis never answers `true` for a verification task that contains a violation of the specification. This favorable characteristic of the value analysis is owed to its over-approximating nature.

[5]`http://www.sosy-lab.org/research/phd/loewe/#PlainValueAnalysis`

	BitVectorsReach	ControlFlow	DeviceDriversLinux64	ECA	Floats	Loops	ProductLines	Sequentialized	Simple	Overall
total	48	48	2 120	1 140	81	141	597	62	46	4 283
unsolved	2	6	1 021	552	0	30	0	10	4	1 625
solved	46	42	1 099	588	81	111	597	52	42	2 658
correct	25	42	1 058	588	38	61	597	39	23	2 471
true	14	27	967	254	16	20	332	8	1	1 639
false	11	15	91	334	22	41	265	31	22	832
incorrect	21	0	41	0	43	50	0	13	19	187
true	0	0	0	0	0	0	0	0	0	0
false	21	0	41	0	43	50	0	13	19	187

Table 3.2: Table showing the verification effectiveness of the value analysis

Incorrect False Verdicts The flip side of the value analysis' over-approximating nature is that it reports numerous incorrect `false` verdicts, i. e., for a verification task that does not contain a violation of the specification, the value analysis alerts the user of a specification violation it supposedly found in that very same verification task. In the experiment we conducted, in at least 187 instances, i. e., for 18 % of all `false` verdicts, the analysis is too imprecise. We say *at least* 187 *instances* here, because we do not perform any witness checking [16], and hence, in the total of 832 correct `false` verdicts, there could be cases in which the value analysis has supposedly found a bug, but this found bug is not really a bug in the verification task at hand, i. e., for some of these 832 verification tasks, the value analysis might have obtained the correct verdict `false` only by chance.

For sure, this effect of the imprecision of the value analysis is not favorable. As briefly mentioned before, CPAchecker as framework already allows counterexample checks to be performed, and in a later chapter of this thesis a smarter, integrated technique for limiting the number of false alarms is presented (cf. Chapter 6). Before elaborating on that, we turn the attention to the verification tasks the value analysis is unable to solve.

Unsolved Verification Tasks In our experiment, for a total of 1 625 verification task, the value analysis is unable to obtain a result, because it either exceeds the defined CPU time limit, or it runs out of memory. The prime reason why the value analysis fails to provide a verdict for a given verification task is state-space explosion.

	BitVectorsReach	ControlFlow	DeviceDriversLinux64	ECA	Floats	Loops	ProductLines	Sequentialized	Simple	Overall
total	0.72	2.1	260	160	0.080	7.7	2.1	3.0	1.1	440
unsolved	0.50	1.5	260	140	0.	7.5	0.	2.5	1.0	410
solved	0.22	0.64	4.4	14	0.080	0.17	2.1	0.49	0.10	22
correct	0.17	0.64	3.6	14	0.036	0.12	2.1	0.33	0.050	21
true	0.15	0.43	3.1	4.8	0.015	0.049	1.0	0.014	0.0045	9.6
false	0.017	0.21	0.46	8.7	0.021	0.069	1.1	0.32	0.046	11
incorrect	0.045	0.	0.81	0.	0.044	0.052	0.	0.16	0.054	1.2
true	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
false	0.045	0.	0.81	0.	0.044	0.052	0.	0.16	0.054	1.2

Table 3.3: Table showing the verification efficiency of the value analysis

Note that in 1 573 of the total 1 625 cases, the respective verification tasks are from the categories DeviceDriversLinux64 or ECA, respectively. Many verification tasks in this two categories are complex and have a huge state space, and as such, these two categories alone account for 97 % of the unsolved verification tasks.

Verification Efficiency of the Value Analysis

For discussing the verification efficiency of the value analysis, we stick to the just mentioned unsolved verification tasks, because, as detailed in Table 3.3, all verification tasks that have an inconclusive result account for 93 % of the total CPU time. Speaking in absolute numbers, this means that out of the total 440 hours CPU time needed by the value analysis —a whopping 410 hours— are practically just wasted.

This whole situation is also captured by the quantile plot shown in Figure 3.3. By looking at the course of the curve, one notes that for many (supposedly simple) verification tasks, the value analysis works well, as well over a third of the verification tasks can be solved promptly in less than 10 seconds. However, approximately from that point on, the value analysis does not scale well anymore for the remaining verification tasks, and it ultimately runs out of resources for the aforementioned total of 1 625 verification tasks.

For the sake of completeness, we performed the same experiment with a timeout of 1 800 s CPU time, with the effect that only a handful of verification tasks more can be solved, while doubling the wasted verification effort from 410 to over 800 hours.

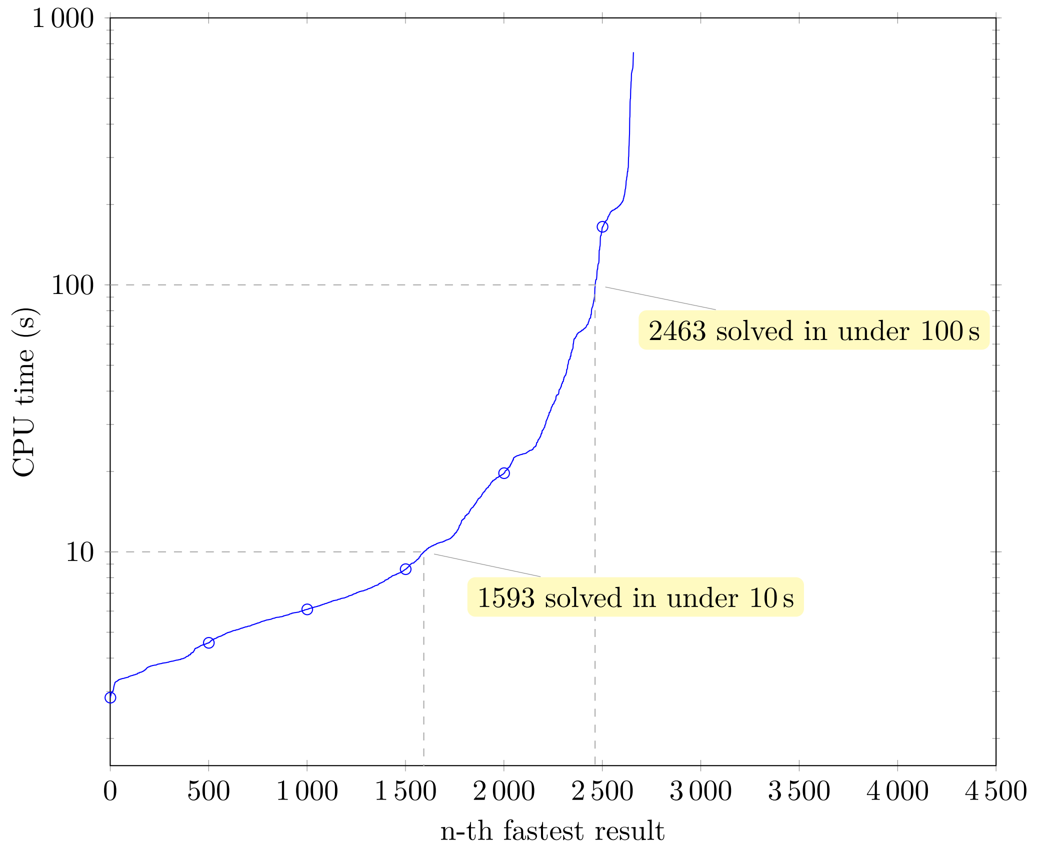


Figure 3.3: The quantile plot for the value analysis, showing that the value analysis is efficient for many verification tasks, but that it does not scale for complex ones

3.6 Conclusion

In this chapter we introduced the value analysis. First, we motivated why it is worthwhile to take a detailed look into such a, at first glance, seemingly simplistic analysis. By an example, we showed how it is supposed to work, and after discussing related work, we formalized the value analysis as CPA, and, based on the implementation in CPAchecker, we performed a thorough evaluation.

3.6.1 Lessons Learned

From that evaluation we learned, that the value analysis performs well on a wide range of benchmarks. From the standpoint of verification effectiveness, it returns a verdict for over 62 % of the verification tasks. The value analysis does not report an incorrect `true` verdict, but in every sixth case where it supposedly found a bug, this "bug" represents a false alarm, i. e., it is an incorrect `false` verdict. Additionally, for more complex verification tasks this simple value analysis fails to provide a meaningful result, because it runs out of resources.

So, the bottom line is that the value analysis is well suited for simple verification tasks and verification tasks where the reachable state space is limited.

3.6.2 Challenge

One challenge to overcome is the imprecision of the value analysis. However, we regard that as a direct consequence of the design decision that, in a given abstract variable assignment, the analysis does only hold the valuation of variables which have a *deterministic* value, while never tracking any information symbolically. After all, this would be more or less what symbolic execution does, and same as for the value analysis, if blindly thrown at more complex verification tasks, approaches based on symbolic execution also fall prey to state-space explosion [75, 78, 101].

So, as first challenge, we try to tackle state-space explosion for the value analysis, more so, because we think that any advances there can also be adopted by other domains or approaches that share similarities with the value analysis, like for example the just mentioned symbolic-execution approach.

3.6.3 Proposition

The current instantiation of the value analysis tracks for all variables their known valuations, independent of the fact whether or not the valuations of a variable are needed for obtaining the verdict for a verification task. Our hypothesis is that, if we had an oracle that could tell the analysis which variables are relevant to reason about a verification task, and the analysis would only track exactly those variables, then this would often reduce the size of the state space dramatically.

3.6.4 Solution

In Chapter 2 we briefly introduced CEGAR and interpolation in the context of software verification, both techniques which are being used with great success for analyses based on predicate abstraction.

Therefore, in order to mitigate the problem of state-space explosion for the value analysis, we propose to incorporate an abstract–refine loop into the value analysis. This will be the main topic of the following chapter.

4 Value Analysis with CEGAR and Interpolation

The previous chapter introduced the value analysis, and defined it as component CPA in the CPA framework. Based on an implementation of the value analysis in the verification framework CPAchecker, we showed that it performs well on a wide range of benchmarks, especially on verification tasks where the reachable state space is only small to medium-sized.

For example, the value analysis solves all verification tasks from the category ProductLines, and also performs well for the verification tasks in the categories ControlFlow and Sequentialized. However, for more complex verification tasks, for example from the category DeviceDriversLinux64, the value analysis often runs out of resources because of state-space explosion, and in the following we pursue ways to avoid exactly this problem.

4.1 Motivation

To see how the value analysis can suffer from state-space explosion, even for a tiny verification task, consider the example verification task shown in Figure 4.1.

This verification task contains a **while** loop in which a generic system call is executed, modelled by the call to the procedure system_call, which is defined as **extern**. The **while** loop exits if either the system call returns 0 or a previously specified number of iterations, given by the value of the program variable **int** x, have been performed. Because the body of the function system_call is unknown, so is the value of the program variable **int** result. Also, because no valuation for **int** x is known, i. e., it also exposes non-determinism, the assumption [ticks > x] cannot be evaluated to either `true` or `false`. This verification task has the verdict `true`, i. e., the **assert** statement in line 18 is not reachable. However, the value analysis as proposed in the previous chapter (cf. Section 3.3), which always tracks every program variable, would keep unrolling the loop $\begin{tikzcd} L8 \arrow[r] & L12 \arrow[r] & L14 \arrow[ll, bend left] \end{tikzcd}$ over and over, continuously discovering new abstract states, because the expression ticks = ticks + 1 always assigns a new value to the program variable **int** ticks. Thus, due to extreme

```
 1 #include <assert.h>
 2 extern int system_call();
 3
 4 int main(int x) {
 5   int flag = 0, ticks = 0;
 6   int result;
 7
 8   while(1) {
 9     ticks  = ticks + 1;
10     result = system_call();
11
12     if(result == 0 || ticks > x) {
13       break;
14     }
15   }
16
17   if(flag > 0) {
18     assert(0);
19   }
20 }
```

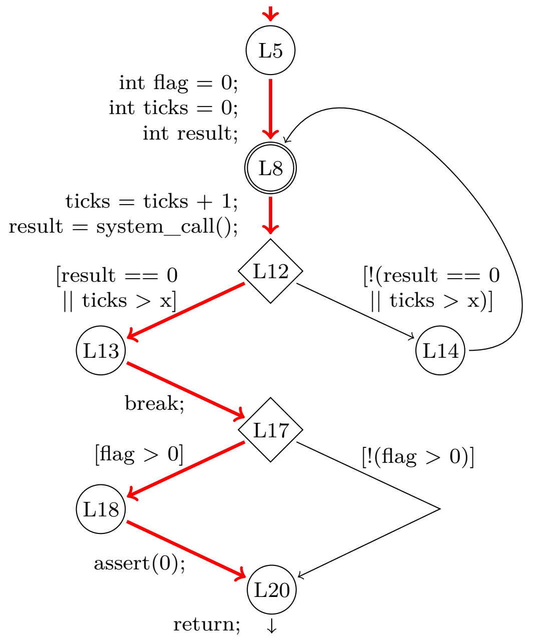


Figure 4.1: Simple verification task, containing an unbounded loop, with its corresponding CFA

resource consumptions, the analysis would not terminate within practical time and memory limits, and, eventually, it is forced to give up on proving the verdict `true` for this verification task.

The new approach for the value analysis that we propose in this chapter can efficiently prove the verdict `true` of this verification task, because it tracks only values of program variables that are necessary to refute the infeasible error paths. In this case only the program variable `int flag` needs to be tracked in order to prove the verdict `true` for this example verification task.

This is achieved by the following approach. In the first iteration of the CEGAR algorithm, the precision of the analysis is empty, i. e., no program variables are tracked. Thus, the `assert` statement will be reached, via the error path shown in the figure, and indicated by the edges that are drawn in red color. Now, using an interpolation technique for the value domain, we can discover a precision from this infeasible error path. In this case, the algorithm identifies that only the program variable `int flag`, or more precisely, the constraint `[flag = 0]` has to be tracked by the state-space exploration algorithm, so that the found error path can be refuted. Therefore, the state-space exploration algorithm is started anew after this refinement, now with the program variable `int flag` in the precision. Because `int ticks` is *not*

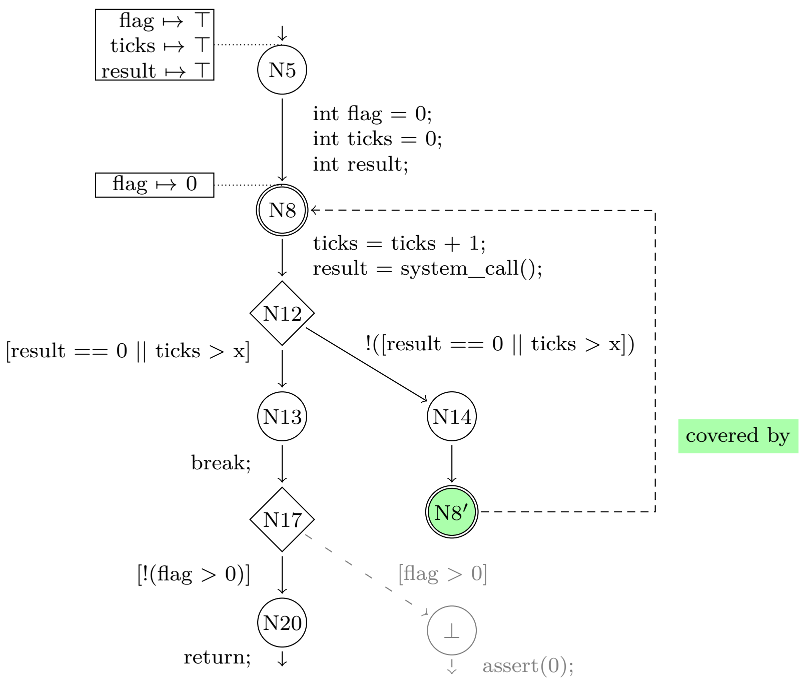


Figure 4.2: The ARG of the verification task from Figure 4.1, annotated with the abstract variable assignments like the value analysis with CEGAR would compute them for the verification task

in the precision, no valuations for this program variable are tracked, and repeated assignments for `ticks = ticks + 1` will not result in new abstract states. This can be seen from Figure 4.2, where the **while** loop is not unrolled. This is the case, because the successor of node N14, namely N8', is covered, i. e., subsumed by the loop head N8, because N8 and N8' both refer to the same location (i. e., the head of the **while** loop), and contain the same abstract variable assignment, namely `{flag` $\mapsto 0$`}`. Therefore the state-space exploration algorithm can stop computing successors for N8'. Finally, the assume operation `[flag > 0]` is evaluated to `false`, thus, the **assert** statement is not reachable, and the analysis terminates, returning the verdict `true`.

In summary, the key effect of this approach is that only relevant program variables are tracked by the value analysis, while unimportant information is ignored. This greatly reduces the amount of information to be tracked, which increases the chance for coverage between abstracts, and as a consequence, reduces the number of abstract states to be explored and thus often avoids the problem of state-space explosion (cf. Figure 4.2). Interestingly, despite the success of abstraction, CEGAR, and interpolation in the field of automatic software verification, these techniques have not been combined and applied together to value domains. In order to fill this gap, we need to define the following components (cf. Section 2.5):

1. the state-space exploration algorithm for the value domain,
2. the precision for the value domain, i. e., the level of abstraction in the value domain,
3. the feasibility check for the value domain, which decides if an error path is feasible according to the abstract semantics of the value domain, and
4. the refinement procedure for the value domain, to refine the precision of the abstract model.

These are the components that are required to leverage the value analysis defined before (cf. Chapter 3), making it applicable in the CEGAR framework, and in the following, after discussing related work in this field, the up-coming sections will introduce the respective components one by one.

4.2 Related Work

The explicit-state model checkers Spin [71], Java Path Finder [66] and Divine [8] were already introduced briefly in the previous chapter (cf. Section 3.2). There exists no work that integrates abstraction based on CEGAR into Divine, and neither does such an approach exist for Spin. But for Spin there exist techniques [72, 73] to extract from a C program a Promela model which represents an abstraction of the original C program. There, the level of abstraction, i. e., the information of what should be included in the Promela model, is determined by extraction rules that the user has to define *manually*. This can be combined with approaches that abstract the Promela model itself [59].

Java Path Finder [66] is designed only for the Java programming language. There exists work that integrates a concept inspired by CEGAR into Java Path Finder, using an approach different from interpolation [94]. In addition, the Bandera tool set [45] can be used to perform slicing of Java programs, and its abstraction engine provides support for data abstraction.

The tool Dagger [62] verifies C programs by applying interpolation-based refinement to octagon and polyhedra domains. To avoid imprecision stemming from the standard widening operator employed in abstract interpretation analyses, Dagger introduces a so called interpolated widening operator, which helps to increase the precision of the analysis and thus avoids false alarms. However, because the interpolated widening operation is in general not monotone, it can happen that a counterexample that is thought to be eliminated by a refinement might be found again in a later stage of the analysis [62].

The algorithm Vinta [2] applies interpolation-based refinement to abstract domains based on intervals. If the state-space exploration finds an error path, then Vinta performs a feasibility check using bounded model checking, and if the error path is infeasible, it computes interpolants. The interpolants are used to refine the invariants that the abstract domain operates on. Vinta requires an SMT solver for feasibility checks and interpolation.

4.3 State-Space Exploration Algorithm for the Value Domain

Because the refinement component as we define it here is independent from the state-space exploration of the value analysis (cf. Chapter 3), we do not need to define a new component here. The only change to be made is, that the precision-adjustment operator prec now does perform an actual abstraction computation, i. e., $\mathsf{prec}(v, \pi, R) = (v_{|\pi}, \pi)$, which means that the abstract variable assignment v is restricted to the set of program variables contained in π, while all others are discarded.

We call this set of program variables the precision π, and the concept of a precision for the value domain will be explained in the following.

4.4 Precision for the Value Domain

The *precision for an abstract variable assignment* is a set π of program variables, and the *value abstraction* for an abstract variable assignment v and a precision π is an abstract variable assignment v' that is defined only on variables that are in the precision π. For example, the value abstraction for the abstract variable assignment $v = \{x \mapsto 2, y \mapsto 5\}$ and the precision $\pi = \{x\}$ is the abstract variable assignment $v' = v_{|\pi} = \{x \mapsto 2\}$. The value abstraction for an abstract variable assignment can be computed using the prec operator defined in Section 3.4.

With the definitions of a precision for abstract variable assignments and value abstraction, we can implement an abstraction scheme that incorporates the two ingredients of lazy abstraction [69].

First, we define the *precision for a program* as a function $\pi : L \mapsto 2^X$, such that for each program location a precision for abstract variable assignments is assigned individually. Thus, instead of tracking variables throughout the whole program, they are only tracked in those parts of the program where they are relevant. The interpolation algorithm introduced below is capable of deriving the information which program variables have to be tracked at which program locations.

The second ingredient of lazy abstraction states that during the analysis of a program different precisions for different abstract states on different program paths in the ARG are being employed. We can apply a *lazy value abstraction*, because again, the interpolation algorithm introduced below allows us to identify the first abstract state (called pivot state [69]) at which novel information is available. Therefore, after a refinement the state-space exploration can continue at this pivot state instead of scheduling a complete recomputation of the state space from scratch.

The value analysis with CEGAR and interpolation uses the precision for a program along with lazy value abstraction in order to compute the value abstraction for an abstract variable assignment at location l by using the precision $\pi(l)$ available at the respective abstract state. The initial program precision for the CEGAR algorithm is the empty program precision $\pi_{init}(l) = \emptyset$, such that the precision-adjustment operator abstracts all program variables, i. e., for each program location $l \in L$, no program variable is tracked. Consequently, if a target location l_e is syntactically reachable in a program, a target state will be reached in the initial CEGAR iteration, due to the initial, empty program precision.

In the following, the component for the feasibility check is introduced, whose task it is then to decide, whether or not the abstract error path also represents a concrete error path.

4.5 Feasibility Check for the Value Domain

The feasibility check for an abstract error path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_e) \rangle$ is performed by running a value analysis, using the full precision $\pi(l) = X$ for all locations $l \in L$, on the path σ. This is equivalent to computing $\widehat{\mathsf{SP}}_{\gamma_\sigma}(\top)$ and check if the result is contradicting, i. e., if $\widehat{\mathsf{SP}}_{\gamma_\sigma}(\top) = \bot$. This feasibility check is extremely efficient, because the abstract error path is finite and the strongest post-operations for abstract variable assignments are simple arithmetic evaluations. If the feasibility check reaches the target state, i. e., $\widehat{\mathsf{SP}}_{\gamma_\sigma}(\top) \neq \bot$, then this abstract error path corresponds to a real counterexample, and the bug can be reported. If the feasibility check does not reach l_e, i. e., $\widehat{\mathsf{SP}}_{\gamma_\sigma}(\top) = \bot$, this means that a refinement is necessary, because the current precision employed by the analysis when finding the abstract error path was insufficient to exclude this infeasible, abstract error path.

In such a case, it is necessary to find a (as small as possible) precision increment, that together with the current precision is sufficient to exclude the current abstract error path in future state-space explorations.

4.6 Interpolation for the Value Domain

The third component needed for embedding the value analysis in the CEGAR framework is a refinement procedure, which will be gradually introduced in the following.

4.6.1 Interpolation for Abstract Variable Assignments

For each infeasible abstract error path in the above mentioned refinement procedure, a precision needs to be determined that assigns to each program location on that path the set of program variables that the value analysis needs to track in order to eliminate that infeasible abstract error path. Interpolation has been proven successful in the predicate domain, however, mind that interpolants from the predicate domain, which consist of formulas, are not useful for the value domain. Hence, we need to introduce the concept of value-domain interpolants, and as a first step we introduce the notion of an interpolant for abstract variable assignments.

An *interpolant* for a pair of abstract variable assignments v^- and v^+, such that $v^- \wedge v^+$ is contradicting, is an abstract variable assignment $\mathcal{V}$ that fulfills the following requirements:

1. the implication $v^- \implies \mathcal{V}$ holds,
2. the conjunction $\mathcal{V} \wedge v^+$ is contradicting, and
3. $\mathcal{V}$ only contains program variables in its definition range which are in the definition ranges of both v^- and v^+, i. e., $\mathsf{def}(\mathcal{V}) \subseteq \mathsf{def}(v^-) \cap \mathsf{def}(v^+)$.

Lemma 4.6.1. *For a given pair (v^-, v^+) of abstract variable assignments, such that $v^- \wedge v^+$ is contradicting, an interpolant exists. Such an interpolant can be computed in time $\mathcal{O}(m + n)$, where m and n are the sizes of v^- and v^+, respectively.*

Proof. The abstract variable assignment $v^-_{|\mathsf{def}(v^+)}$ is an interpolant for the pair of abstract variable assignments (v^-, v^+). □

The above-mentioned interpolant that simply results from restricting v^- to the definition range of v^+, i. e., force a common definition range for v^- and v^+, is of course not the best interpolant. Moreover, interpolation over abstract variable assignments is not expressive enough, as the following example illustrates.

Consider the constraint sequence $\gamma = \langle i = 0, i > 0 \rangle$, which could occur as (part of an) infeasible, abstract error path. Note that $\gamma = \langle i = 0, i > 0 \rangle$ is contradicting, because $\widehat{\mathsf{SP}}_\gamma(\top) = \bot$. However, there exists no interpolant for the pair of abstract variable assignments $v^- = \widehat{\mathsf{SP}}_{\langle i=0 \rangle}(\top) = \{i \mapsto 0\}$, and $v^+ = \widehat{\mathsf{SP}}_{\langle i>0 \rangle}(\top) = \top$, because the conjunction of the two abstract variable assignments v^- and v^+ is not contradicting, i. e., $v^- \wedge v^+ = \{i \mapsto 0\} \wedge \top = \{i \mapsto 0\} \neq \bot$.

Hence, interpolation for abstract variable assignments is a first idea to approach interpolation in the value domain, but since interpolants need to be extracted for paths, in the following, we define interpolation for constraint sequences.

4.6.2 Interpolation for Constraint Sequences

A more expressive interpolation is achieved by considering constraint sequences. In order to define interpolation for constraint sequences, we introduce definitions for conjunction, implication and contradicting for constraint sequences.

We define the *conjunction* $\gamma \wedge \gamma'$ of the constraint sequences $\gamma = \langle op_1, \ldots, op_m \rangle$ and $\gamma' = \langle op'_1, \ldots, op'_n \rangle$ as their concatenation, i. e., $\gamma \wedge \gamma' = \langle op_1, \ldots, op_m, op'_1, \ldots, op'_n \rangle$, the *implication* of γ and γ' (denoted by $\gamma \implies \gamma'$) as the implication of their strongest post-conditions $\widehat{\mathsf{SP}}_\gamma(\top) \implies \widehat{\mathsf{SP}}_{\gamma'}(\top)$. Furthermore, we say that a constraint sequence γ results in a *contradiction*, if $\widehat{\mathsf{SP}}_\gamma(\top) = \bot$, i. e., $\widehat{\mathsf{SP}}_\gamma(\top)$ is contradicting.

An *interpolant* for a pair of constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting, is a constraint sequence Γ that fulfills the three requirements:

1. the implication $\gamma^- \implies \Gamma$ holds,
2. the conjunction $\Gamma \wedge \gamma^+$ is contradicting, and
3. Γ contains in its constraints only program variables that occur in the constraints of both γ^- and γ^+.

Lemma 4.6.2. *For a given pair (γ^-, γ^+) of constraint sequences, such that $\gamma^- \wedge \gamma^+$ is contradicting, an interpolant exists. Such an interpolant is computable in time $\mathcal{O}((m+n)^2)$, where m and n are the sizes of γ^- and γ^+, respectively.*

Proof. Algorithm Interpolate (cf. Algorithm 3) returns an interpolant for two constraint sequences γ^- and γ^+. The algorithm starts with computing the strongest post-condition for γ^- and assigns the result to the abstract variable assignment v, which then may contain up to m program variables. Per definition, the strongest post-condition for γ^+ of the abstract variable assignment v is contradicting. Next, we perform a *value-domain interpolation query*, or an *interpolation query* for short. In such an interpolation query we test for a program variable from v if after removing it from v the strongest post-condition for γ^+ of v is still contradicting, and we do such an interpolation query for each program variable from v (each such test takes n $\widehat{\mathsf{SP}}_{op}(\cdot)$ steps). If it is still contradicting, the program variable can be removed from v. If not, the program variable is necessary to prove the contradiction of the

Algorithm 3: Interpolate(γ^-, γ^+)

```
Input     : two constraint sequences $\gamma^-$ and $\gamma^+$, with $\widehat{\mathsf{SP}}_{\gamma^- \wedge \gamma^+}(\top) = \bot$
Output    : a constraint sequence $\Gamma$, which is an interpolant for $\gamma^-$ and $\gamma^+$
Variables : an abstract variable assignment $v$
1 $v := \widehat{\mathsf{SP}}_{\gamma^-}(\top)$;
2 foreach $x \in \mathsf{def}(v)$ do
3     if $\widehat{\mathsf{SP}}_{\gamma^+}(v_{|\mathsf{def}(v) \setminus \{x\}}) = \bot$ then
          // $x$ is not relevant and should not occur in the interpolant
4         $v := v_{|\mathsf{def}(v) \setminus \{x\}}$;
  // start assembling the interpolating constraint sequence
5 $\Gamma := \langle\rangle$;
6 foreach $x \in \mathsf{def}(v)$ do
      // append an assume constraint for $x$
7     $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;
8 return $\Gamma$
```

two constraint sequences, and thus, should occur in the interpolant. Note that this keeps only program variables in v that occur in γ^+ as well. The rest of the algorithm constructs a constraint sequence from the abstract variable assignment, in order to return an interpolating constraint sequence, which fulfills the three requirements of an interpolant. With this algorithm such an interpolant can be computed in time $\mathcal{O}((m+n)^3)$. $\square$

Mind that, if we would remove the loop starting in line 2 of Algorithm 3 and instead just restrict v to those program variables being referenced in $\widehat{\mathsf{SP}}_{\gamma^+}(v)$, Algorithm 3 would return a valid interpolant in time $\mathcal{O}((m+n)^2)$. However, in general this would lead to interpolants being too strong. So, we check for each program variable $x \in \mathsf{def}(v)$ individually its influence on the feasibility of $\widehat{\mathsf{SP}}_{\gamma}(\cdot)$, by issuing an interpolation query. Hence, we are investing more resources during interpolation, so that we can save computational effort during the main analysis —and often avoid divergence— because we are able to employ a coarser precision.

4.7 Refinement Based on Value Interpolation

The goal of our interpolation-based refinement for the value analysis is to determine a location-specific precision that is strong enough to eliminate an infeasible error path in future state-space explorations. This criterion is fulfilled by the property of interpolants. A second goal is to have a precision that is as weak as possible,

Algorithm 4: Refine(σ)

```
Input     : infeasible error path $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$
Output    : precision $\pi$
Variables : interpolating constraint sequence $\Gamma$
1 $\Gamma := \langle\rangle$
2 foreach $l \in L$ do
3     $\pi(l) := \emptyset$
4 for $i := 1$ to $n - 1$ do
5     $\gamma^+ := \langle op_{i+1}, ..., op_n \rangle$
      // inductive interpolation
6     $\Gamma :=$ Interpolate($\Gamma \wedge \langle op_i \rangle, \gamma^+$)
      // extract variables from variable assignment resulting from $\Gamma$
7     $\pi(l_i) := \{x \mid (x, z) \in \widehat{\mathsf{SP}}_\Gamma(\top)\}$
8 return $\Pi$
```

i. e., to have a precision with a definition range as small as possible, in order to be parsimonious in tracking program variables and creating abstract states.

We apply the idea of interpolation for constraint sequences to assemble the precision-extraction algorithm Refine (Algorithm 4). It takes as input an infeasible error path, and returns a precision for a program. A further requirement is that the procedure computes *inductive* interpolants [18], i. e., each interpolant along the path contains enough information to prove the remaining path infeasible. This is needed in order to ensure that the interpolants at the different locations achieve the goal of providing a precision that eliminates the infeasible error path from further explorations. For every program location l_i along an infeasible error path σ, starting at l_0, we split the constraint sequence of the path into a constraint prefix γ^-, which consists of the constraints from the start location l_0 to l_i, and a constraint suffix γ^+, which consists of the constraints from the location l_i to l_e. For computing inductive interpolants, we replace the constraint prefix by the conjunction of the last interpolant and the current constraint. The precision is extracted by computing the abstract variable assignment for the interpolating constraint sequence and assigning the relevant program variables as precision for the current location l_i, i. e., the set of all program variables that are necessary to be tracked in order to eliminate the error path. This algorithm can be directly plugged in as refinement routine of the CEGAR algorithm (cf. Algorithm 2). Also note that the interpolation and refinement depends only on $\widehat{\mathsf{SP}}_{op}(\cdot)$, which is the same operator used in the main analysis. This self-containedness is another nice property of our novel approach.

The interpolation and refinement process is illustrated in Figure 4.3, using the

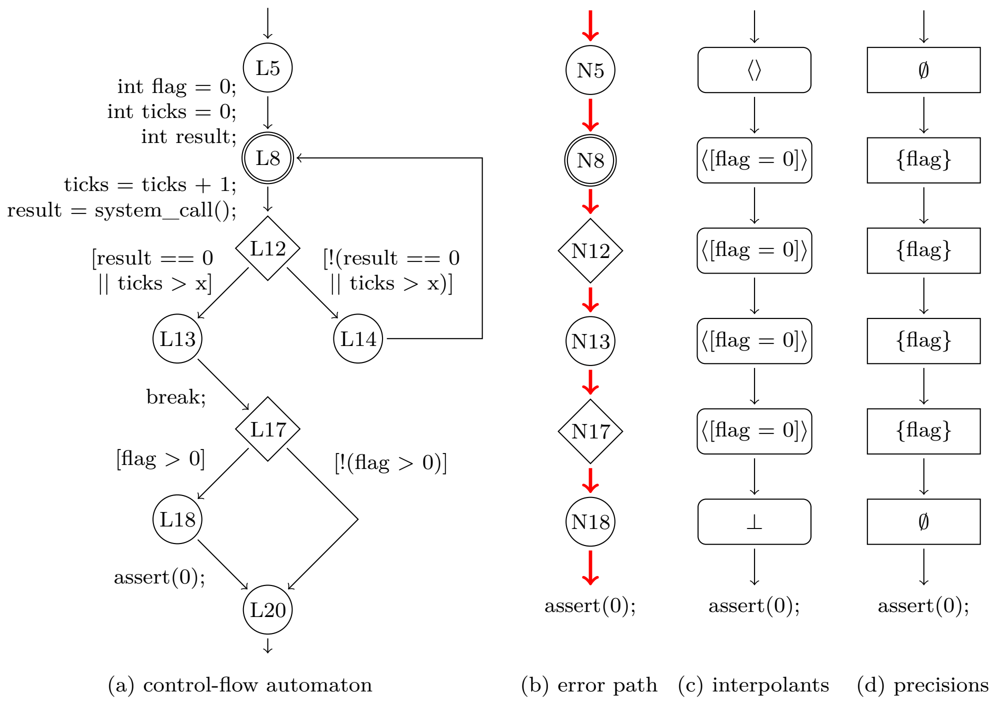


Figure 4.3: For the introductory example from Figure 4.1 the (a) CFA is shown, the (b) infeasible error path the value analysis will find if started with an empty precision, the (c) sequence of interpolants as constraint sequence, and the (d) precision elements as extracted from the interpolants needed to refute the infeasible error path from future state-space explorations

introductory example from this chapter (cf. Figure 4.1). By default, the value analysis with CEGAR starts the state-space exploration for every verification task with an empty precision, and consequently, for the verification task in Figure 4.3a the error path in Figure 4.3b will be found, eventually. This path is then checked for feasibility, and it turns out to be an infeasible error path. Therefore, it is passed to the procedure Refine (cf. Algorithm 4) which then calls procedure Interpolate (cf. Algorithm 3) to compute the interpolants (cf. Figure 4.3c) for each program location of the infeasible error path, before, back in procedure Refine, the respective precision elements (cf. Figure 4.3d) are extracted from each of the interpolants. The resulting precision is strong enough to prove the verdict `true` for the verification task in the next iteration of the CEGAR loop. Mind that the precision remains abstract enough, such that the loop in the verification task is not unrolled, and the verification process of this verification task concludes promptly.

Based on the concepts above, we implemented a refinement component for the value domain, and in the following a thorough evaluation of this approach is presented.

4.8 Evaluation

For evaluating the value analysis, now with CEGAR and interpolation, we use the same experimental setup and the same benchmark verification tasks as described in Sections 3.5.1 and 3.5.2, respectively.

4.8.1 Configuration

In order to guarantee that the results in this evaluation stay comparable to the results we obtained from the evaluation of the value analysis *without* CEGAR (cf. Section 3.5) we use the same revision 20 406 of CPAchecker, and we again advise the analysis to not descend into recursive function calls and to not double-check on any counterexamples, just like before. We use BenchExec the same way as before for executing the benchmarks.

The configuration corresponding to the value analysis with CEGAR is named `valueAnalysis-Cegar`, and it configures the analysis to respect the two main ideas of lazy abstraction [69]. This means that after a refinement this configuration makes the analysis continue from the pivot state, i. e., the parent state of the state closest to the initial state for which in the current refinement new precision elements were found. This way, only those parts in the state space that really need the new precision actually get the new precision, while all other parts of the state space are still explored with a coarser precision, which ultimately might lead to a smaller state space. Second, this configuration advises the analysis to make use of a *parsimonious precision*, i. e., it only tracks program variables at those program locations where the refinement procedure identified that their valuations are required for excluding an infeasible counterexample, while not tracking them anywhere else, which, again, ultimately might lead to a smaller state space.

In order to allow reproducibility of the evaluation, an example for a complete command line to run the value analysis with CEGAR as well as the full results and raw data are available on our supplementary web page [1].

4.8.2 Results

We now present the results of running the value analysis with CEGAR on the benchmarks and in the environment as described before (cf. Section 3.5).

Verification Effectiveness of the Value Analysis with CEGAR

Here we discuss in detail only a few interesting aspects regarding the verification effectiveness of the value analysis with CEGAR. For a complete overview of the

[1] `http://www.sosy-lab.org/research/phd/loewe/#ValueAnalysisCegar`

	BitVectorsReach	ControlFlow	DeviceDriversLinux64	ECA	Floats	Loops	ProductLines	Sequentialized	Simple	Overall
total	48	48	2 120	1 140	81	141	597	62	46	4 283
unsolved	2	3	750	1 140	1	25	207	34	2	2 164
solved	46	45	1 370	0	80	116	390	28	44	2 119
correct	25	45	1 311	0	37	63	390	22	25	1 918
true	14	27	1 148	0	15	22	262	8	2	1 498
false	11	18	163	0	22	41	128	14	23	420
incorrect	21	0	59	0	43	53	0	6	19	201
true	0	0	0	0	0	0	0	0	0	0
false	21	0	59	0	43	53	0	6	19	201

Table 4.1: Table showing the verification effectiveness of the value analysis with CEGAR and interpolation

characteristics regarding the verification effectiveness of the value analysis with CEGAR on our benchmark set, we point the reader to Table 4.1, and especially to the discussion in Section 4.8.3, where the differences between running the value analysis with and without CEGAR are highlighted.

Solved Verification Tasks In total, the value analysis with CEGAR solves 2 119 out of the 4 283 verification tasks. This means, that for almost one half of the verification task —for 49 % to be exact— the value analysis with CEGAR obtains a result of either `true` or `false`. It fares quite well in all categories, except for the category `ECA` where it is unable to solve even a single verification task.

Incorrect True Verdicts The value analysis with CEGAR also does not return a single incorrect `true` verdict across the whole benchmark set, i. e., the refinement component, as one would expect, does not make the analysis unsound in that regard.

Verification Efficiency of the Value Analysis with CEGAR

For the verification efficiency of the value analysis with CEGAR, we believe that mere numbers about the consumed run time alone have little value, so we rather discuss this in the next section, where we compare the value analysis with CEGAR to the value analysis without CEGAR, to which we refer from now on as the *plain* value analysis.

4.8.3 Comparison to the Plain Value Analysis

In the beginning of this chapter, we showed how the plain value analysis can easily fall prey to the problem of state-space explosion, and we claimed that incorporating CEGAR into the analysis, this would improve both the verification effectiveness and the verification efficiency of the analysis.

We will now check this claim by comparing both the verification effectiveness and the verification efficiency of the plain value analysis to that of our novel value analysis with CEGAR and interpolation.

Differences in Verification Effectiveness

To show the differences in verification effectiveness of the two approaches, we present in Table 4.2 the cell-wise difference of Table 4.1 and Table 3.2, respectively. So, a positive value in a cell of Table 4.2 means that the value analysis with CEGAR achieves a higher result for the respective table cell as the plain value analysis, and vice versa. Accordingly, a cell value of 0 means, that both approaches obtain the same number of results for the respective category.

From Table 4.2 one can see, that for half of the categories, namely, for the categories `BitVectorsReach` `ControlFlow`, `Floats`, `Loops`, and `Simple` it does not make a big difference whether to use this CEGAR approach or not, because over these five categories, the approach with CEGAR solves in total a mere 9 verification tasks more.

For the category `DeviceDriversLinux64`, the CEGAR approach performs nicely, as 271 verification tasks can be solved, which remain unsolved by the plain value analysis. The CEGAR approach works well for more instances in this category, because, while they contain a large number of program variables, often only few of them are needed to reason about the verdict of the verification task, and our interpolation technique is able to identify these. The plain value analysis cannot make this distinction, but instead tracks each and every program variable, which blows up the state space and renders the verification infeasible in many cases.

For the category `ECA`, the CEGAR approach fails, not solving a single verification task. These verification tasks do not contain many relevant program variables, however, they have an extremely complex control flow containing thousands of nested assume operations within an endless loop. What we observe here is also related to state-space explosion, or more precisely, to path explosion, because a precision has to be computed for every single error path that is found during the analysis of such a verification task. Due to the high branching rate in this class of verification

	BitVectorsReach	ControlFlow	DeviceDriversLinux64	ECA	Floats	Loops	ProductLines	Sequentialized	Simple	Overall
total$_\triangle$	48	48	2 120	1 140	81	141	597	62	46	4 283
solved$_\triangle$	0	3	271	−588	−1	5	−207	−24	2	−539
correct$_\triangle$	0	3	253	−588	−1	2	−207	−17	2	−553
true$_\triangle$	0	0	181	−254	−1	2	−70	0	1	−141
false$_\triangle$	0	3	72	−334	0	0	−137	−17	1	−412
incorrect$_\triangle$	0	0	18	0	0	3	0	−7	0	14
true$_\triangle$	0	0	0	0	0	0	0	0	0	0
false$_\triangle$	0	0	18	0	0	3	0	−7	0	14

Table 4.2: Table showing the difference in verification effectiveness between the plain value analysis and the value analysis with CEGAR

tasks, a huge number of different abstract error paths exist, each of which has to be refuted in its own interpolation-based refinement step. In the end, the search for a precision that is strong enough proves to be too costly for these verification tasks, and the verification process runs out of time. The degradations in the categories ProductLines and Sequentialized can also be explained by this effect, because the verification for verification tasks in these categories also runs into a timeout after hundreds of refinements that do not yield a suitable precision.

In total, the value analysis with CEGAR solves 539 verification tasks fewer than the plain value analysis. A prime reason why the CEGAR approach fails can be drawn from the scatter plot shown in Figure 4.4. The plot puts in relation the time needed to verify a verification task using the plain value analysis to the time needed to verify the same verification task using the value analysis with CEGAR. The color of a data point tells how many refinements are necessary to verify the respective verification task using the value analysis with CEGAR, i. e., a deep blue data point signals that no more than 100 refinements are needed, and a red data point signals that 1 000 refinements and more are performed during the course of the analysis. For the visualization we omit data points for those verification tasks that both analyses cannot solve, i. e., where both approaches run out of resources.

In this plot there are three clusters distinguishable. One, in a blueish hue in the lower left corner. This cluster represents tasks that can be solved efficiently by both approaches. This cluster is shifted a bit above the diagonal because of the overhead that the CEGAR approach has to invest for performing refinements —

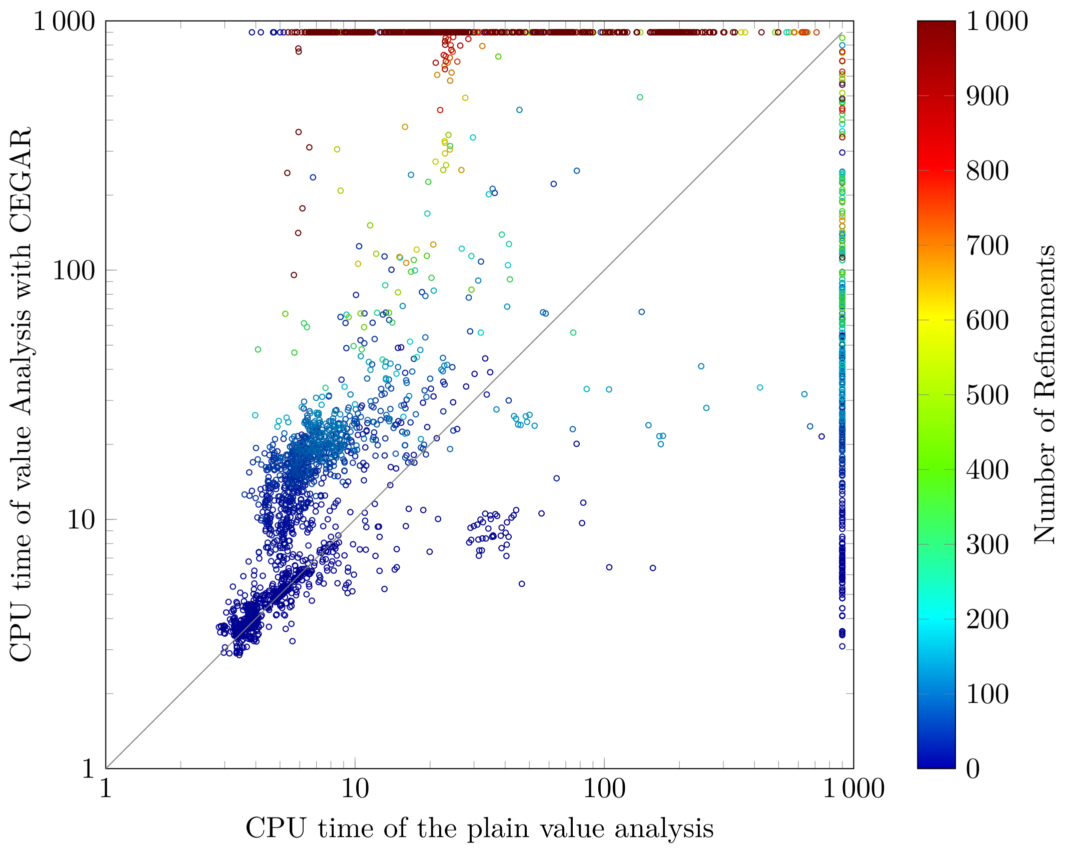


Figure 4.4: CPU time for the plain value analysis versus CPU time for the value analysis with CEGAR, with the color of the data points indicating the number of refinements performed during the CEGAR approach

even for simpler tasks. Another cluster can be seen on the far right border of the plot, representing those 329 verification tasks where the CEGAR approach pays off, because only the value analysis with CEGAR is able to solve these —mostly with relatively few refinements, indicated by the blueish to greenish hue. Finally, the last cluster is on the top border of the plot. These 868 data points refer to verification tasks that only the plain value analysis can solve. With the high number of reddish data points in that cluster, the scatter plot provides evidence that the value analysis with CEGAR performs poorly on these verification task because of the huge number of refinements that are performed during the analysis.

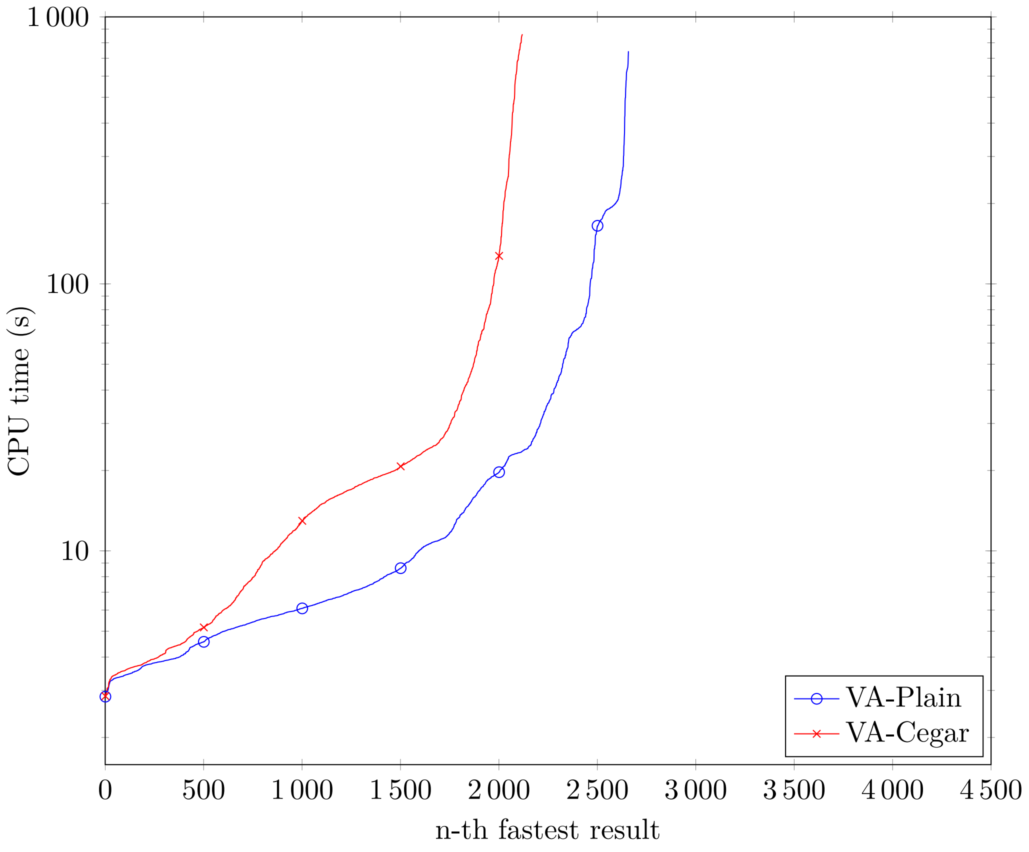


Figure 4.5: The quantile plot for the value analysis with CEGAR, and for the plain value analysis, where the latter performs strictly better

Differences in Verification Efficiency

For discussion of the verification efficiency of the value analysis with CEGAR, we refer to the quantile plot shown in Figure 4.5. The graph indicates clearly, that this CEGAR approach is not yet ready for prime time. Not only ends the graph for the value analysis with CEGAR more to the left than the one for the plain value analysis —the lower verification effectiveness was already discussed— but it also runs above the graph for the plain value analysis, which means it is also less efficient. In total, the value analysis with CEGAR needs around 570 hours to run on all 4 283 verification tasks. This is an increase of over 30 % when compared to the 440 hours that the plain value analysis needed.

4.9 Conclusion

In this chapter we introduced an interpolation-based refinement component for the value analysis. Based on an example, we showed how a CEGAR approach is able to circumvent the problem of state-space explosion occurring in the plain value analysis presented before (cf. Chapter 3). We then defined a precision, a feasibility check, and a refinement procedure for the value domain, that together with the state-space exploration algorithm of the plain value analysis form a complete analysis based on CEGAR. Finally, an evaluation, in which we compared our novel approach to the plain value analysis allowed us to draw some valuable conclusions on which we reflect in the following.

4.9.1 Lessons Learned

In the course of this chapter, we gained insights on how to design a CEGAR component that is applicable to the value domain. We were able to incorporate the main ideas of lazy abstraction [69], and an evaluation proved that our first approach to enable CEGAR in the value domain allows the successful verification of verification tasks that the plain value analysis cannot solve. However, for now, we still face the challenge, that the number of verification tasks the CEGAR approach can solve is lower than what the plain value analysis can solve.

4.9.2 Challenge

During the evaluation presented in the previous section it became apparent, that the low performance of the value analysis with CEGAR seems to be linked to the high number of refinements that the CEGAR approach performs in search for a precision that is strong enough. In addition, during each such refinement, new interpolants have to be computed, which often requires tens or even hundreds of thousands interpolation queries to be performed for a single verification task, which eventually renders the verification infeasible.

4.9.3 Proposition

The main reasons that slow down the CEGAR approach are the costly interpolation queries, and the high number of refinements. So, we need to lower the overhead imposed by the refinement component in order to make the value analysis with CEGAR competitive. The procedure Interpolate, as presented in Section 4.6.2, is kept simple and straight-forward, and we propose to investigate potential optimizations to reduce the computational cost. Furthermore, we will try to reduce the number of refinements by generating stronger precisions so that one refinement excludes more potential abstract error paths at once.

4.9.4 Solution

In order to speed up the interpolation process, heuristics can be implemented that still return valid interpolants, but with less interpolation queries. For reducing the number of refinements, the model checker Blast adds predicates —its precision elements— not only at the locations dictated by the interpolant, but also adds them at a wider scope in order to avoid similar, repeated refinements [18]. Besides that, other techniques to reduce the overhead of the CEGAR approach are worth to be investigated, e. g., a technique we refer to as *global refinement*. There a refinement is not started immediately for every target state, but several abstract error paths are collected first, before then a single refinement computes interpolants for all collected error paths at once, with the hope that the overall process converges faster.

5 Value Analysis with Improved CEGAR and Interpolation

In the course of the previous chapter a refinement component for the value analysis was defined and evaluated. While the evaluation identified a few strong points of our novel approach, it also revealed several shortcomings, which disallow the value analysis with CEGAR to be effective and efficient on a wider set of verification tasks. Therefore, in this chapter, we will focus on various techniques to improve the verification effectiveness and verification efficiency of the CEGAR approach for the value analysis.

5.1 Motivation

The value analysis with CEGAR performs slightly better in the SV-COMP categories ControlFlow, Loops, and Simple, and solves almost 300 verification tasks more in the category DeviceDriversLinux64 than the plain value analysis does, which alone is reason enough to not give up on the CEGAR approach for the value domain, but rather try to improve it, such that it surpasses the plain value analysis, and becomes a competitive analysis — on our whole benchmark set, and beyond. The first challenge we take up is to reduce the high number of value interpolation queries that are performed during each refinement step, and in the following we present several techniques that greatly help to achieve that goal. We evaluated these techniques in the same experimental setup as described before (cf. Section 3.5), and we refer to the results throughout this section.

5.2 Reducing the Number of Value Interpolation Queries

The procedure Interpolate, presented in Algorithm 3, is a costly operation for itself, and during the refinement of one infeasible error path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$, in the worst case, this procedure has to be called up to n times, once for each operation along the path. In order to give an idea of the extent of potential savings, in the evaluation we performed to compare the different optimizations for the interpolation process, the value analysis with CEGAR and all optimizations for the interpolation process disabled needs around 24 hours to solve 2 120 verification tasks. The interpolation process alone needs 6 hours —around 25 % of the total time— in which an enormous amount of close to 30 million interpolation queries are performed. Note that none of these interpolation optimizations affect the strength of the resulting interpolants, i. e., they do not become stronger, as the number of refinements stays the same, no matter if using none or all optimization techniques at once (cf. Table 5.1).

5.2.1 Iterative, Inductive Interpolation

The refinement step has to ensure the availability of a sufficient precision at each program location along an infeasible error path σ. To achieve this, an interpolant has to be computed at each offset i along the path $\sigma = \langle(op_1, l_1), \ldots, (op_n, l_n)\rangle$, with $0 < i < n$. Other than perhaps indicated by the signature of $\mathsf{Interpolate}(\gamma^-, \gamma^+)$ (cf. Algorithm 3), it is not the case that at the i-th offset, and at the i-th call to Interpolate, the abstract variable assignment v in $v := \widehat{\mathsf{SP}}_{\gamma^-}(\top)$ is computed from the constraint sequence $\langle op_1, \ldots, op_i\rangle$. Instead, there γ^- often corresponds only to a small fraction of $\langle op_1, \ldots, op_i\rangle$, in fact, only the *relevant* fraction. This can actually be seen in the procedure Refine (cf. line 6 in Algorithm 4), where an inductive interpolant is computed by calling Interpolate with $\Gamma \wedge \langle op_i\rangle$ as the first parameter, i. e., the next interpolant is computed from the current interpolant Γ conjugated with the current operation op_i, and the suffix γ^+.

Arguably, the computation of inductive interpolants is not really an optimization, but rather a requirement to obtain a valid precision for excluding an infeasible error path. However, it leads to a less costly computation of $v := \widehat{\mathsf{SP}}_{\gamma^-}(\top)$ in Interpolate, because the sequence γ^- usually is far shorter than $\langle op_1, \ldots, op_i\rangle$, as γ^- only contains the relevant constraints of $\langle op_1, \ldots, op_i\rangle$.

5.2.2 Interpolation over Deepest Infeasible Suffix

An actual optimization that allows the procedure Interpolate to work more efficiently comes from the insight, that Interpolate can only compute non-empty, or non-trivial interpolants for the *deepest infeasible suffix* of an error path $\sigma = \langle(op_1, l_1), \ldots, (op_n, l_n)\rangle$, where the deepest infeasible suffix $\sigma^\perp$ of σ is defined as $\sigma^\perp = \langle(op_i, l_i), \ldots, (op_n, l_n)\rangle$, with $\widehat{\mathsf{SP}}_{\sigma^\perp}(\top) = \perp$ and $\widehat{\mathsf{SP}}_{\langle(op_{i+1}, l_{i+1}), \ldots, (op_n, l_n)\rangle}(\top) \neq \perp$.

To help better understand this optimization, consider the loop starting at line 2 of procedure Interpolate (cf. Algorithm 3). There we check via an interpolation query for each variable individually, if its valuation is needed to make the suffix γ^+ contradicting. Consequently, if the suffix γ^+ is contradicting in itself, i. e., if $\widehat{\mathsf{SP}}_{\gamma^+}(\top) = \perp$ holds, then no variable valuations are needed to ensure the infeasibility of the suffix γ^+ and Interpolate would return the empty, i. e., the trivial, constraint sequence, but not before needlessly performing interpolation queries.

So by construction of the procedure Interpolate, what actually is being interpolated during the course of a refinement is not the complete error path σ, but instead only the deepest infeasible suffix $\sigma^{\perp}$. Therefore, we can introduce a short-cut that first checks if $\widehat{\mathsf{SP}}_{\gamma^+}(\top) = \bot$ holds, and if so, return the empty constraint sequence as interpolant right away. Assuming that the abstract variable assignment v in $v = \widehat{\mathsf{SP}}_{\gamma^-}(\top)$ would contain assignments for n variables, this optimization saves, in each call to Interpolate where $\widehat{\mathsf{SP}}_{\gamma^+}(\top) = \bot$ holds, $n - 1$ interpolation queries.

For similar reasons, no further calls to Interpolate are necessary if $\widehat{\mathsf{SP}}_{\Gamma \wedge \langle op_i \rangle}(\top) = \bot$ holds, i. e., once the current prefix is contradicting, because the contradiction is reached in the original infeasible error path σ, after which no precision, and hence, no interpolants are needed anymore.

For the 2 120 verification tasks solved, this optimization alone safes over 5 million interpolation queries.

5.2.3 Interpolant-Equality Heuristic

The heuristic based on equality of the current interpolant and a candidate interpolant is another technique to avoid costly, and also needless interpolations.

For understanding the approach of this heuristic, assume that, before the call to Interpolate (cf. line 6 in Algorithm 4), we have the current interpolant Γ for the offset op_i. We know that we can obtain an interpolant that is valid also up to the next operation op_{i+1} by computing $v = \widehat{\mathsf{SP}}_{\Gamma \wedge \langle op_{i+1} \rangle}(\top)$ and directly assemble the next interpolant Γ' from v, i. e., we call Interpolate but skip the loop (cf. line 2 in Algorithm 4) and hence, the weakening of the candidate interpolant by not performing any interpolation queries. This interpolant can never be too weak, but it could potentially be stronger than needed. However, as experiments show, if we limit this heuristic to cases where the candidate interpolant Γ' equals the current interpolant Γ, i. e., it did not become stronger for this operation op_{i+1}, then it pays off to immediately return Γ' as next interpolant and effectively save all interpolation queries in this call to Interpolate.

To explain this heuristic in more detail, consider the following example. Assume that for the call to Interpolate the current interpolant Γ is non-trivial, e. g., lets assume it equals the constraint sequence $\langle$`[var = 0]`$\rangle$ over a global variable `int var`. Let us further assume the next operation op_{i+1} is the assignment of the global variable `int bar` to the value of another global variable `int foo`, i. e., `bar = foo`, both which are not referenced in the current interpolant Γ. In the first line of Interpolate, we would then compute v as $v := \widehat{\mathsf{SP}}_{\langle \texttt{[var = 0]} \rangle \wedge \langle \texttt{bar = foo} \rangle}(\top)$, which evaluates to the abstract variable assignment $v = \{\texttt{var} \mapsto 0\}$, and, according to the heuristic, we could assemble and return the interpolant $\Gamma' = \langle \texttt{[var = 0]} \rangle = \Gamma$, without performing any interpolation queries.

Note that for this example, no interpolation queries are needed, and the interpolant Γ' returned by the heuristic is the best possible. This is because the only interpolants we can assemble from v are the empty constraint sequence and the constraint sequence $\langle$`[var = 0]`$\rangle$. The empty constraint sequence cannot be a valid interpolant here, because this would mean that $\widehat{\mathsf{SP}}_{\gamma^+}(\top) = \bot$ holds, i. e., γ^+ is contradicting in itself, thus, the trivial, empty interpolant would suffice, which however cannot be the case, because for the previous operation op_i, we had the non-empty interpolant $\langle$`[var = 0]`$\rangle$. Therefore, the only possible interpolant is $\Gamma' = \langle$`[var = 0]`$\rangle = \Gamma$.

So, in conclusion, whenever the interpolant directly assembled from v equals the current interpolant, then we can reuse the current interpolant. This heuristic always yields a valid interpolant, and, as experiments show, this heuristic practically always returns the same interpolant as procedure Interpolate, but at far lower computational cost.

For the 2 120 verification tasks solved, this optimization alone safes almost 14 million interpolation queries.

5.2.4 Interpolant-Equivalence Heuristic

Furthermore, when interpolating an error path, there is another optimization possible, e. g., in the case where parameters of function are part of the interpolant.

Again, consider the global variable **int** var is needed for refuting an infeasible error path, and let us assume that the current interpolant holds the constraint `[var = 0]`. Let us further assume that the operation op_i for which to compute the next interpolant is the call to the function var = foo(var), for which the signature is **int** foo(**int** bar). In this case, we do not have to perform an actual interpolation, but can simply remove the constraint `[var = 0]` for the current interpolant and replace it by the constraint `[foo::bar = 0]` [1], because what basically happens here is a renaming from var to foo :: bar.

Due to technical reasons, in the implementation similar renamings are performed when processing **return** statements or when a function jumps back to the call site.

For the 2 120 verification tasks solved, this optimization alone safes over one million interpolation queries.

5.2.5 Evaluation of the Optimizations for the Value Interpolation

In order to allow reproducibility of the evaluation, an example for a complete command line to run the value analysis with the optimizations presented above as well as the full results and raw data are available on our supplementary web page [2]

[1]Namespaces, as done with prefix foo for variable bar, are needed for inter-procedural analyses.
[2]`http://www.sosy-lab.org/research/phd/loewe/#valueAnalysisCegarItpOptimizations`

	Time for Interpolation (h)	Number of Refinements	Number of Itp-Queries	Reduction of Itp-Queries
NoOptimizations	5.9	153 496	29 164 794	✗
DeepestInfeasibleSuffix	4.3	153 496	23 867 772	5 297 022
InterpolantEquality	3.3	153 496	14 930 702	14 234 092
InterpolantEquivalence	5.6	153 496	27 706 464	1 458 330
AllOptimizations	1.8	153 496	9 607 674	19 557 120

Table 5.1: Overview of the different optimization techniques and how they relate to not applying any optimizations and applying all optimizations

In Table 5.1 we summarize the savings achieved by the respective optimizations. Note that it is possible to apply all optimizations together at once. The result of this can be seen in the row named AllOptimizations. When applying all optimizations at once, the number of interpolation queries can be reduced to around 33 % if compared to the non-optimized variant. A similar reduction rate is achieved for the interpolation time, which decreases from almost 6 hours down to under 2 hours.

Despite these positive results, only 20 verification tasks more can be solved if applying the combination of all optimizations. This indicates that the interpolation technique is not the real bottleneck, and so in the next section, we now investigate on ways to reduce the number of refinements, i. e., we try to get the CEGAR approach to converge faster.

5.3 Reducing the Number of Refinements

In Section 3.4.2 we introduced the concept of a precision for the value analysis, and later in Section 4.4 we concretized the precision for the value domain, which is defined as a function $\pi : L \mapsto 2^X$, that maps from locations to program variables. One of the main ideas of lazy abstraction [69] is the use of a parsimonious, or localized precision, i. e., the mapping from program locations to precision elements —in case of the value analysis, these are program variables— is only defined for those program locations where the interpolation procedure identified precision elements to be required for excluding an infeasible error path.

While this approach keeps the precision coarse, it may also force the analysis to enumerate all abstract error paths one by one, if the relevant precision elements are spread over different program paths and locations. This fits well our experience in case the value analysis is configured to use a localized precision, as we then often notice that consecutive refinements extract precisions containing the same set of program variables, but with each refinement they are associated with different program locations.

In order to avoid this effect for the value analysis, we propose to extend the range of the precision elements, i. e., the program variables, to the respective scope of the program variables. Mind that a similar optimization is also implemented in the software model checker Blast [69]. In contrast to the normal localized precision, the result is, what we call, a *scoped* precision. So, in case the interpolation procedure identifies a global variable to be relevant at any location, the resulting precision will advise the analysis to track this global variable throughout verification task. Similar for local variables, if the interpolation procedure identifies a local variable to be relevant at any location within a function, the resulting precision will signal the analysis to track this local variable at all program locations belonging to the function.

Mind that a scoped precision can turn out to be disadvantageous for the analysis, because program variables are tracked at program locations where they actually do not need to be tracked, potentially adding again to the problem of state-space explosion. So, it is important that there is a good balance between the reduction of refinements and the overhead of tracking superfluous information.

As it turns out, if we run the value analysis with CEGAR using the scoped precision instead of the localized, parsimonious precision, as advocated by the lazy abstraction principle, then we are able to reduce the number of refinements significantly. In our evaluation we achieved a reduction of 85 % —from 145 007 down to only 20 910 refinements— for the 2 087 verification tasks that both approaches can solve. At the same time there are hardly any negative effects from tracking superfluous information, as there are only 32 verification tasks that are exclusively solved using the localized precision, while there are 729 verification tasks that are exclusively solved using the scoped precision.

In summary, the switch from the localized precision to the scoped precision greatly improves the performance of the overall analysis, on which we report in more detail during the next section.

5.4 Evaluation

In this section, we again evaluate the value analysis with CEGAR, but now with improved CEGAR and interpolation, i. e., we run it with the optimizations for the interpolation procedure, and with the scoped precision introduced above. In order to allow drawing comparisons with the previous evaluations from Sections 3.5

and 4.8, respectively, we reuse the same benchmark verification tasks in the identical experimental setup, including the use of BenchExec just as in the previous evaluation sections.

5.4.1 Configuration

We again take CPAchecker in revision 20 406, and provide the main configuration file `valueAnalysis-Cegar-Optimized`, along with the options to skip descending in recursive function calls and not performing counterexample checks, same as in the evaluations before.

Besides applying the optimizations from Sections 5.2 and 5.3, we also choose to restart the analysis from the initial state with the new precision after a refinement, i. e., we *fully* disable lazy abstraction. While for most categories it hardly makes a difference whether to continue from the pivot state or restart the analysis after a refinement, our experiments show, that this is especially beneficial for the categories `ECA` and `ProductLines`. This is the case, because both these classes of verification tasks have a rather complex control flow with a high degree of branching, and the program variables identified as relevant by a refinement are not only relevant in a couple of sub-trees of the ARG, but are relevant throughout the verification task. Therefore, for these verification tasks, it is more efficient to add them to the precision at the top-level once.

In order to allow reproducibility of the evaluation, an example for a complete command line to run the value analysis with improved CEGAR and interpolation as well as the full results and raw data are available on our supplementary web page[3].

5.4.2 Results

We now present the results of running the value analysis with improved CEGAR and interpolation in the same experimental setup as in the previous evaluations. For a complete overview of the results obtained by this instance of the value analysis, we refer the reader to Table 5.2. From there one can see, that the value analysis with improved CEGAR and interpolation now can solve 72 %, or 3 088 of the 4 283 verification tasks. This means, that, due to the improvements discussed in this chapter so far, the value analysis with improved CEGAR and interpolation can solve a total of 969 verification tasks more than without the improvements.

In terms of verification effectiveness, the biggest improvement is achieved in the category `ECA`, where now 587 verification tasks can be solved, while without the improvements, none are solved. Other categories, where the verification effectiveness is now considerably improved are `DeviceDriversLinux64`, `ProductLines`, and

[3]`http://www.sosy-lab.org/research/phd/loewe/#ValueAnalysisCegarPlus`

	BitVectorsReach	ControlFlow	DeviceDriversLinux64	ECA	Floats	Loops	ProductLines	Sequentialized	Simple	Overall
total	48	48	2 120	1 140	81	141	597	62	46	4 283
unsolved	2	3	452	553	1	27	137	20	0	1 195
solved	46	45	1 668	587	80	114	460	42	46	3 088
correct	25	45	1 583	587	37	63	460	31	27	2 858
true	14	27	1 390	287	15	22	283	8	3	2 049
false	11	18	193	300	22	41	177	23	24	809
incorrect	21	0	85	0	43	51	0	11	19	230
true	0	0	0	0	0	0	0	0	0	0
false	21	0	85	0	43	51	0	11	19	230

Table 5.2: Table showing the verification effectiveness of the value analysis with improved CEGAR

Sequentialized such that now the value analysis with improved CEGAR and interpolation clearly out-performs the plain value analysis in terms of verification effectiveness, as shown in the next section.

5.4.3 Comparison to the Plain Value Analysis

We now present a comparison between the plain value analysis and the value analysis with the now improved approaches for CEGAR and interpolation.

Similar as in Section 4.8.3, we show the differences in verification effectiveness of the two approaches in a table representing the cell-wise difference of Table 5.2 and Table 3.2, respectively.

From the resulting table (cf. Table 5.3) we can see that the value analysis with CEGAR is now clearly more effective than the plain value analysis, as the approach based on CEGAR can solve 430 verification tasks more. The biggest improvement coming from the optimizations detailed in the previous section is due the good results in the category ECA— remember that without the optimizations no verification task can be solved there. Also, on the category DeviceDriversLinux64, the improved approaches for CEGAR and interpolation allow even better results, as now 569 verification tasks can be solved that the plain value analysis cannot solve.

The results in the category ProductLines remain the only blemish of the value analysis with CEGAR, as a total of 137 verification tasks less can be solved when

	BitVectorsReach	ControlFlow	DeviceDriversLinux64	ECA	Floats	Loops	ProductLines	Sequentialized	Simple	Overall
total	48	48	2 120	1 140	81	141	597	62	46	4 283
solved$_\triangle$	0	3	569	−1	−1	3	−137	−10	4	430
correct$_\triangle$	0	3	525	−1	−1	2	−137	−8	4	387
true$_\triangle$	0	0	423	33	−1	2	−49	0	2	410
false$_\triangle$	0	3	102	−34	0	0	−88	−8	2	−23
incorrect$_\triangle$	0	0	44	0	0	1	0	−2	0	43
true$_\triangle$	0	0	0	0	0	0	0	0	0	0
false$_\triangle$	0	0	44	0	0	1	0	−2	0	43

Table 5.3: Table showing the difference in verification effectiveness between the plain value analysis and the value analysis with improved CEGAR

relying on the value analysis with CEGAR. An explanation for this effect is given in this next section (cf. Section 5.4.4).

In order to compare the verification efficiency of the plain value analysis with that of the value analysis with and without improved CEGAR and interpolation, we refer to the quantile plot shown in Figure 5.1.

The graph corresponding to the value analysis with improved CEGAR and interpolation is the lowest of all, which tells us that this analysis is the most efficient for solving the verification tasks in our benchmark set. Only at the far left, there are a few verification tasks where the plain value analysis is a tiny bit faster. This is due to the overhead that the CEGAR approach comes with for finding a suitable abstraction first, but it is surprising that this overhead is basically compensated. The value analysis with improved CEGAR and interpolation is also clearly better suited for harder verification tasks, because the graph corresponding to the plain value analysis goes almost straight up for tasks needing more than 100 s, while the graph corresponding to the value analysis with improved CEGAR and interpolation tends further to the right, i. e., it solves more of the harder tasks as already discussed above.

5.4.4 Level of Non-Determinism

So far in this chapter we showed that a CEGAR approach for the value analysis is able to out-perform the plain value analysis in terms of verification efficiency. Still,

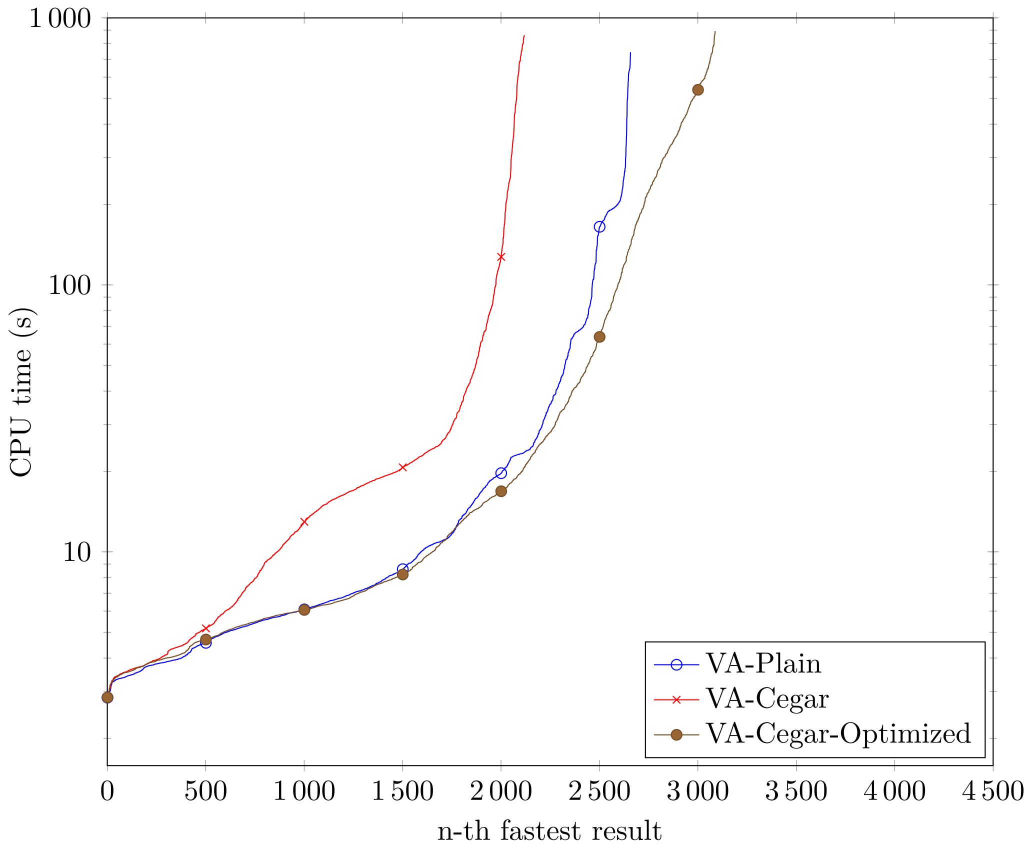


Figure 5.1: The quantile plot for the plain value analysis, for the value analysis with CEGAR, and for the value analysis with improved CEGAR, where the latter performs best

for some verification tasks, the CEGAR approach does not pay off, which seemingly is true for large parts of category PRODUCTLINES (cf. Table 5.3). However, we argue that it is not the case that the value analysis with CEGAR performs bad there, but that it is rather the case, that the plain value analysis performs extremely well there. We back this claim by referring to, what we call, the *level of non-determinism* of a verification task, and we compute the level of non-determinism for a verification task as follows. We run the value analysis *without* CEGAR, counting the total number of branching nodes, i. e., assume operations like [a == 1] or [a >= b], that are traversed during the analysis. The analysis also takes note how many of the assume operations have two successors states. Mind that if the plain value analysis computes two successors for an assume operation, this indicates that the assume operation could not be evaluated to a deterministic value, i. e., non-determinism is present in the respective verification task. This fact is of interest here, because if there exists a successor for both the `then` branch as well as for the **`else`** branch of an assume operation, then this means that two independent branches need to be explored by the analysis, and if that happens often, then the state space of the verification task may become huge. In contrast, if most assume operations only allow at most a single successor, than the state space of the verification task is limited, it even may degenerate to a path program if each and every assume operation can be evaluated deterministically. Interestingly, the latter is the case for many verification tasks in the category ProductLines, and therefore, verification tasks from this category can be solved easily by the plain value analysis, as it tracks as much information as possible without ever performing abstraction computations.

In contrast, the value analysis with CEGAR does not perform particularly well in the category ProductLines. To give an intuition why this is the case, lets assume several refinements were performed already and only a subset of all relevant program variables is being tracked. Because not all program variables referenced in assume operations are being tracked, there is a lot of case splitting going on, i. e., both branches of an assume operation have to be explored in many cases. But at the same time, because program variables are already being tracked, and these have different valuations in different branches, there is also less chance for coverage when two independent branches meet again. So now there exist many independent branches that end in states where the variable valuations are different, so coverage is not possible and each branch must be explored individually which eventually leads to state-space explosion. Furthermore, the verification tasks in the category ProductLines require many program variables to be tracked in order to be able to exclude all infeasible error paths, and the effort spent for the continuous computation of refinements also hinders the value analysis with CEGAR to perform well here.

The scatter plot in Figure 5.2 provides evidence for our claims, covering verification tasks of the categories ECA and ProductLines, that the plain value analysis or the value analysis with CEGAR is able to solve. In this plot, the x-coordinate of a data point corresponds to the CPU time needed to solve the respective verification task with the plain value analysis, and the y-coordinate corresponds to the CPU time needed to solve the very same verification task with the value analysis with CEGAR. The color of the data point reflects the level of non-determinism —the measure we briefly sketched above— which ranges from 0 % to 100 %. A low level of non-determinism, i. e., close to 0 % is drawn in a bluish hue, while a high level of non-determinism is drawn in a reddish hue. The plot makes two points rather clear. First, the verification tasks in the ECA and ProductLines expose a rather low level of non-determinism, as practically all data points are colored blue. Second, the plain value analysis is better suited for almost all these verification tasks, as only a few data points are located in the lower triangle below the diagonal, while many are positioned close to border on the top, indicating that the plain value analysis can

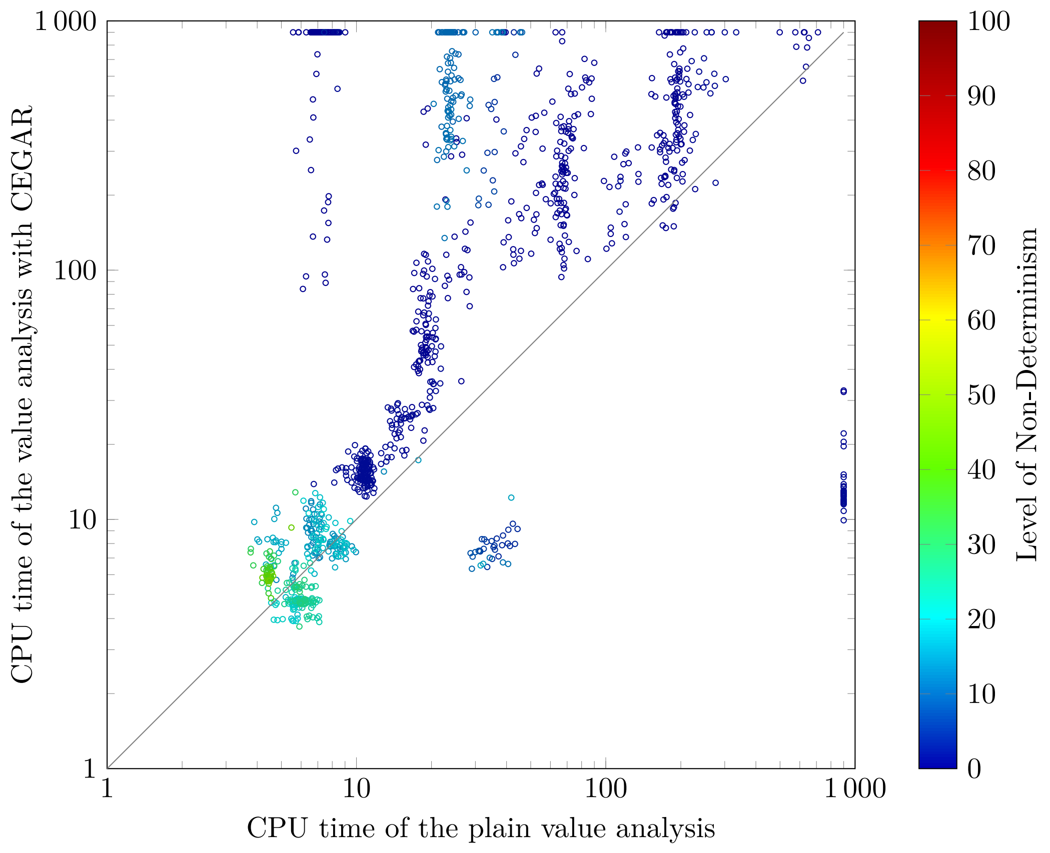


Figure 5.2: Scatter plot comparing the CPU time of the plain value analysis (x-axis) to the CPU time of the value analysis with CEGAR (y-axis) over the verification tasks in the categories `ECA` and `ProductLines`, with the color of the data points indicating the level of non-determinism of the respective verification task

solve the respective verification tasks, but that the value analysis with CEGAR runs out of time when trying to solve them.

In contrast, the plot in Figure 5.3 covers verification tasks from all categories *except* for `ECA` and `ProductLines`, and here now, the level of non-determinism is particularly higher, as witnessed by the high number of data points with a reddish hue. Interestingly, for verification tasks exposing a high level of non-determinism, the value analysis with CEGAR is superior over the plain value analysis, which can be seen from the massive number of reddish data points along the right border of the plot, which represent verification tasks with a higher level of non-determinism that the value analysis with CEGAR can solve but the plain value analysis cannot. To give an intuition why this is the case, consider a verification task that is highly

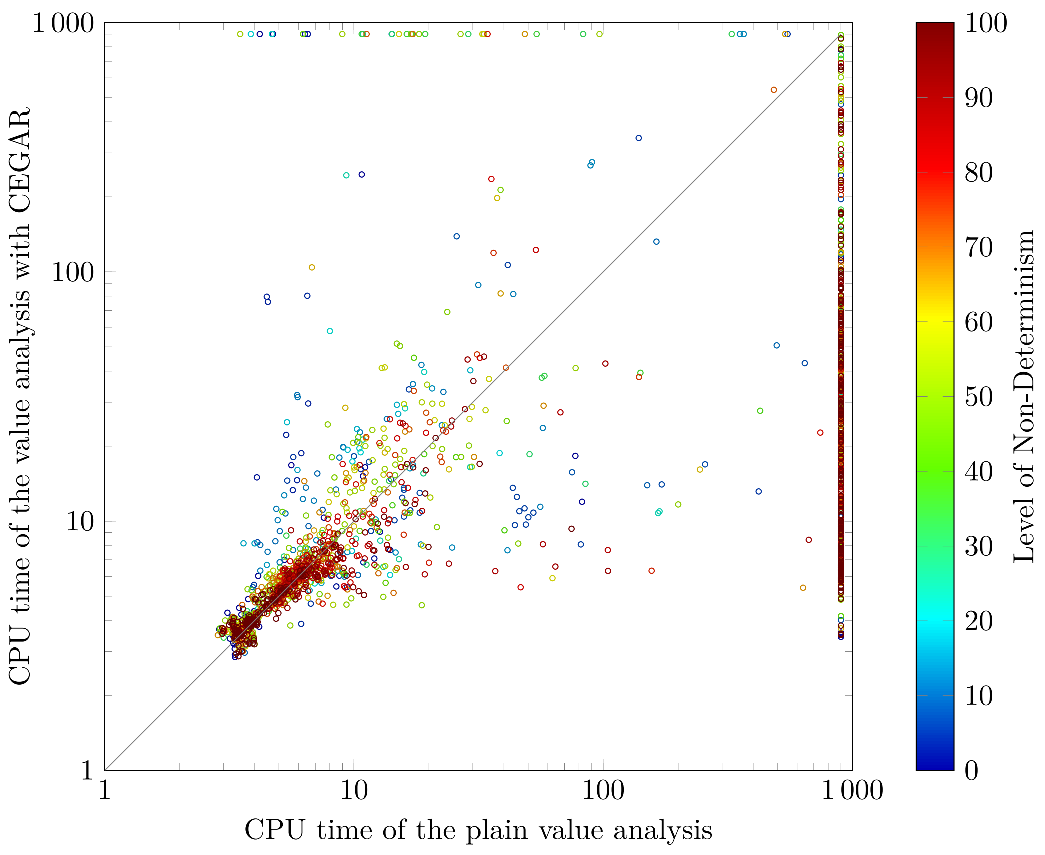


Figure 5.3: Scatter plot comparing the CPU time of the plain value analysis (x-axis) to the CPU time of the value analysis with CEGAR (y-axis) over all verification tasks except those from the categories `ECA` and ProductLines, with the color of the data points indicating the level of non-determinism of the respective verification task

non-deterministic, i. e., many assume operations in this verification task cannot be evaluated to a deterministic value, hence, always both paths need to be considered by the value analysis, independent of the fact whether the value analysis relies on CEGAR or not. However, what makes the difference here is that the plain value analysis —running with full precision— tracks all program variables within the different paths, while the value analysis with CEGAR normally only tracks a small fraction of all the program variables. According to our evaluation, on average over the commonly solved tasks, the value analysis with CEGAR tracks less than 25 % of the program variables the plain value analysis tracks. This means that coverage relations are more likely for the value analysis with CEGAR, which reduces the number of paths that have to be explored independently, and ultimately allows the analysis to finish in time.

So, after having established that the value analysis with CEGAR is beneficial for verification tasks with a higher level of non-determinism, while the plain value analysis is best suited for rather deterministic verification tasks, an obvious suggestion would be to combine the two approaches in a composite analysis. A simple approach could first start the plain value analysis, and let it run for, e. g., 100 s as most verification tasks the plain value analysis solves are finished in under 100 s anyway. After these 100 s, the value analysis with CEGAR could then use the remaining time to solve as many verification tasks as possible. However, while this approach is valid, and yields more results than a single analysis alone, it also is not very interesting, because, using the data from our evaluation, we can compute an estimated number of solved verification tasks for this approach by counting each verification task that was either solved by the plain value analysis in under 100 s or by the value analysis with CEGAR in 800 s or less.

A more sophisticated approach for a composite analysis would be to base the decision on which analysis to use on the level of non-determinism that the current verification task exposes, e. g., by delaying the abstraction computation until the level of non-determinism is above a certain threshold, as detailed in a later chapter (cf. Section 9.1). However, while some combination of two value analyses might lead to some improvements, we believe that a composite analysis of two analyses backed by two *different* domains is far more interesting and powerful.

But, before focusing on the composition of the value analysis with a symbolic analysis (cf. Chapter 6), we first discuss some other interesting research directions for the value analysis.

5.5 Versatility of Value-Analysis Refinement

In the previous section we discussed the benefits that our novel refinement component has for the value analysis. In addition, the availability of such a refinement component opens up several more possibilities, and on two of these we will briefly report on.

5.5.1 Applicability to other Analyses

Besides the value analysis presented in this thesis, there exist many other analyses based on various abstract domains, like for example the interval domain [46], which is also rather imprecise but allows an efficient analysis, too, or the polyhedra domain [48], which is more precise but comes with huge memory costs.

The octagon domain [89] is yet another abstract domain, which is aimed to perform, in terms of verification expressiveness and verification efficiency, somewhere in between the interval domain and the polyhedra domain. This octagon domain is based on difference-bound matrices which are able to express constraints of the form $\pm v_1 \pm v_2 \leq c$ with $v_1, v_2 \in X$ and $c \in \mathbb{Z}$. The octagon domain allows to manipulate these constraints with an $\mathcal{O}(n^2)$ worst-case memory cost per abstract state and an $\mathcal{O}(n^3)$ worst-case time cost per abstract operation, with n being the number of program variables [89].

Analyses based on these abstract domains often join abstract states when control flow meets [47], in order to avoid state-space explosion, and to increase the verification efficiency of the analysis. However, this is to the disadvantage of the precision of the analysis, which usually leads to reporting many incorrect `false` verdicts.

The verification framework CPAchecker also holds an implementation of an analysis based on the octagon domain, and there exists work that integrates the value-analysis refinement along with the value interpolation into the octagon analysis. In an evaluation of 2 300 verification tasks the octagon analysis with CEGAR is able to solve 1 800 verification tasks, while the octagon analysis without CEGAR can only solve 1 500 — both using merge_{sep} as merge operator. The memory consumption is reduced to around 50 % on average if the CEGAR approach is used. More importantly, the CEGAR approach also solves clearly more verification tasks than the classic approaches based on joining and widening, while also reporting far less incorrect `false` verdicts [4].

Symbolic execution [25,37,75,78,101] also suffers from state-space explosion. In its original form it tracks a symbolic value for each program variable and explores each program path individually. For a verification task containing n assume operations up to 2^n different paths have be to explored, which often is too expensive for any non-trivial verification task.

The verification framework CPAchecker also has an analysis based on symbolic execution, and there exists an extension that adds interpolation and refinement, similar as it is presented here for the value analysis, to the analysis based on symbolic execution. In an evaluation using the verification tasks of SV-COMP'16, symbolic execution with CEGAR achieves impressive results [25] by solving well over 2 700 verification tasks, while the original approach only solves 921 verification tasks [5].

This shows that the concepts of interpolation and refinement, as introduced in this thesis, are applicable to a whole range of abstract domains, and even to analyses that track the program state symbolically, like symbolic execution.

5.5.2 Regression Verification

In the software industry of today, regression testing is an established and well-understood technique [61, 95]. However, as with any testing approach, regression

[4]The experimental setup was similar as presented throughout this thesis.
[5]The experimental setup was again similar as presented throughout this thesis.

testing is incomplete, i. e., it is not powerful enough to verify that an arbitrary piece of code is free of errors, nor is it capable of exhaustively checking an arbitrary verification task for errors. Augmenting regression testing with regression *verification* [27, 65, 99] seems like a legit idea. However, verifying a single verification task once is already considered costly, especially when compared to testing, and this is magnified in the light of regression verification, where not a single verification task has to be verified once, but many, and this over and over again, from revision to revision. Thus, naive approaches for regression verification are bound to fail in an industrial setting.

But consider this. Any approach based on CEGAR, be it the value analysis with CEGAR, or a predicate analysis, or any other analysis based on CEGAR, all of them have to invest quite some effort first into repeatedly exploring and refining the state space until the abstract model becomes precise enough to be reasoned about undisputedly. For many verification tasks the effort spent up to the final refinement is by far higher than the effort needed after the final refinement. This means that reasoning about an abstract model that reflects the relevant characteristics of the actual verification task just precisely enough is often inexpensive, and naturally it is less of an effort than having to go through the whole process of computing that abstract model over several refinement iterations.

The concept of precision reuse can be exploited for regression verification. The main idea there is to reuse the set of precisions —those that are extracted in the respective refinements during the verification process— for verifying future revisions [27].

With the definition of refinement and precision for the value analysis, plus formalizing an exchange format for writing and reading these precisions, the technique for efficient regression verification based on precision reuse becomes immediately available for the value analysis, too [27]. In an evaluation on an industrial scale, precision reuse for efficient regression verification was evaluated for the value analysis with CEGAR on 4 193 verification tasks, stemming from 62 Linux kernel device drivers spanning over a total of 1 119 revisions. The verification of these 4 193 verification tasks took 13 000 s when verifying each verification task from scratch, but only 4 900 s when using precision reuse, underlining that efficient and robust regression verification is now also available for the value analysis [27].

So, precision reuse may enable efficient regression verification, but note that in case the precision for the initial revision is not suitable for an efficient verification, e. g., because it forces the unrolling of a complex loop, then the verification of future revision will likely fail, too. Therefore, during a refinement, it is crucial to find a suitable precision that neither is too strong nor forces the analysis to unroll loops. We will present such techniques in Chapter 7, but before that, we shed some light on alternative approaches we investigated in our quest for effective verification techniques.

5.6 Further Considerations

Before concluding this chapter, we give a short overview of alternative interpolation and refinement techniques for the value analysis. Some of these approaches were already suggested before in other domains, e. g., for the predicate analysis, and our intuition was that they also perform well for the value domain. While this held to be true to some extent, our studies and evaluation prove that none of these techniques presented below are groundbreaking. Nevertheless, they represent interesting concepts and allow a deeper understanding of the whole matter, especially as it becomes clear why the respective approaches are inferior to the approaches discussed in earlier chapters of this work.

5.6.1 Static Refinement

If refinement based on value interpolation is performed, (cf. Section 4.7) then in order to obtain interpolants for an infeasible error path parts of this infeasible error path are *evaluated* repeatedly, giving this approach a *dynamic* character.

Of course, this interpolation-based approach is not the only way to compute the precision of the value analysis, and in the following we present an idea of computing a new precision by performing a purely static backwards analysis along the infeasible error path, hence, we call it *static* refinement.

Extracting Precisions from Use-Def Chains

This approach is based on the simple idea, that, in order to exclude an infeasible error path, it is sufficient to track all program variables that are referenced in the use-def chains [1, 56, 74] computed over all assume operations.

In Algorithm 5 we present the procedure ObtainInUseFunction. This procedure computes for an infeasible error path σ the *in-use* function υ_σ, which maps from locations $l \in L$ to sets of program variables $\chi \in 2^X$. To extract the program variables being defined or used in an operation $op \in Ops$, the procedure delegates to $\mathsf{defs}(op)$ and to $\mathsf{used}(op)$, respectively. For the simple imperative programming language on which our formalism is based on $\mathsf{defs}(op)$ returns, as singleton set, the program variable on the left hand side of op if op is an assignment operation and the empty set if op is an assume operation. The call $\mathsf{uses}(op)$ returns the set of program variables on the right hand side of op if op is an assignment operation and the set of all program variables in op if op is an assume operation. The in-use function υ is then computed by traversing the path σ once in reverse order. For an assume operation at a location l all program variables occurring in this assume operation are added to the current set χ_{inUse} and then to $\upsilon(l)$. When processing an assignment operation at a location l, it first is checked if the program variable being assigned, i. e., the program variable currently in the set χ_{defs}, is in use, i. e., if it is in χ_{inUse}. Only if that is the case, then the program variable being assigned is removed from the current set χ_{inUse}, and all program variables used in that assignment operation are added to the current set χ_{inUse} and then to $\upsilon(l)$.

A non-empty set $\chi = \upsilon(l)$ for a location l means that the program variables contained in χ are needed to allow the evaluation of an assume operation after the location l. As the path σ is infeasible there exists at least one contradicting assume operation in σ such that $\widehat{\mathsf{SP}}_\sigma(\top) = \bot$ holds, and a precision computed from υ is strong enough to exclude the infeasible error path σ. Transforming the function $\upsilon : L \mapsto 2^X$ to a precision is trivial, because υ already conforms to the definition of a precision, being $\pi : L \mapsto 2^X$.

Comparison to Value Interpolation

The technique presented above is not based on value interpolation. After all, this purely static approach does not deal with abstract variable assignments or constraint

Algorithm 5: ObtainInUseFunction(σ)

```
Input     : an infeasible error path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_e) \rangle$
Output    : the in-use function $\upsilon : L \mapsto 2^X$, mapping from locations $l \in L$ to sets
            of variables $\chi \subseteq X$
Variables : sets of variables $\chi_{defs} \subseteq X, \chi_{uses} \subseteq X, \chi_{inUse} \subseteq X$
1  $\chi_{inUse} = \emptyset$;
2  foreach $l \in L$ do
3      $\upsilon(l) := \emptyset$
4  for $i = n$ to $i = 1$ do
       // set of variables being written in $op_i$
5      $\chi_{defs} := \mathsf{defs}(op_i)$;
       // set of variables being read in $op_i$
6      $\chi_{uses} := \mathsf{uses}(op_i)$;
       // any uses in assume operation always become in-use
7      if isAssumeOperation($op_i$) then
8          $\chi_{inUse} := \chi_{inUse} \cup \chi_{uses}$;
       // uses of def in in-use become in-use
9      else if $\chi_{defs} \cap \chi_{inUse} \neq \emptyset$ then
10         $\chi_{inUse} := (\chi_{inUse} \setminus \chi_{defs}) \cup \chi_{uses}$;
       // add in-use to function $\upsilon$ at location $l_i$
11     $\upsilon(l_i) := \chi_{inUse}$;
12 return $\upsilon$;
```

sequences but only operates on identifiers of program variables, and for these the formalisms like contradiction, implication and others (cf. Section 4.6) are undefined.

Due to the static nature of the approach, it is not known which assume operations along a path actually are contradicting, and so the approach has to consider all assume operations to be contradicting. This is also the reason why precisions computed by procedure ObtainInUseFunction are never weaker, i. e., they always contain at least the program variables that precisions contain which are computed using value interpolation (cf. Algorithm 4).

In the evaluation we performed, again in the same experimental setup as in the sections before, this approach solves 231 verification tasks more than the *plain* value analysis. But compared to the value analysis with CEGAR and interpolation it fails to solve 199 verification tasks that can be solved with the optimized refinement approach from before (cf. Section 5.4). Hence, the extra effort spent for the value interpolation pays off, and due to that insight, we do not try to improve this static refinement approach directly. However, we note that static refinement may profit dramatically from the technique presented in Chapter 7, which allows to modify an infeasible error path prior to computing a (static) refinement in such a way that at most a single contradicting assume operation remains in the respective infeasible error path. Consequently, the static refinement approach does not need to consider all assume operations to be contradicting, but it suffices to compute the use-def-chain starting from the single contradicting assume operation, often allowing a far more concise precision increment.

5.6.2 Global Refinement

Another technique that we studied and that has already been suggested for the predicate domain computes refinements not for a single error path but for multiple error paths at once [3], i. e., for error paths forming a directed acyclic graph. In the classic CEGAR approach a refinement is initiated as soon as the first target state has been identified, and for each such target state being found during the course of the analysis a refinement is performed. When refining a set of paths at once, this is done differently. In the extreme case, the analysis first enumerates —according to the current precision— all reachable states of the current verification task. If no target state is found, the analysis returns the verdict `true`, and if a feasible counterexample is detected, the analysis returns the verdict `false`, just as in the classic CEGAR approach. In all other cases the refinement procedure is tasked to compute a *single* precision increment that is strong enough to exclude *all* infeasible error paths present in the computed state space. We call this *global refinement* and we propose to use that also for our value analysis.

With the value analysis it is often too expensive to always unroll the complete reachable state space between subsequent refinements when analyzing more complex verification tasks, so for our work with the value analysis, we favour a trade-off between the one extreme that always triggers a refinement right after having found a new target state and the other extreme that always unrolls the complete reachable state space before initiating a refinement. To this end, we can configure the value analysis such that a refinement is initiated (1) once a certain amount of target states have been found, or (2) once a certain number of abstract states have been computed after having found the next target state. This allows us to choose any configuration between the two extremes mentioned above.

There are three main benefits that we aim to achieve. First, as multiple error paths are identified between two subsequent refinements, the chance of finding a feasible error path within less refinements is higher, so we hope to find bugs faster. Second, with this approach the number of refinements needed to solve a verification task is expected to be lower compared to the classic CEGAR approach, and therefore the number of re-explorations of the state space are also reduced, potentially speeding up the overall verification process. Third, there are now several infeasible error paths scheduled for interpolation during one refinement step, so synergy effects for the interpolation procedure may emerge.

To explain the last aspect in more detail, we show the source code of an illustrative example verification task on the left in Figure 5.4. This verification task contains four **assert** statements, each representing a target location. Note that all of them are actually unreachable, so the verdict of this verification task is `true`. If we give this verification task as input to the value analysis with CEGAR and enable global refinement, due to the initially empty precision, the analysis reaches all four target locations and identifies all of them as infeasible. On the right side of Figure 5.4, a tree structure shows the interpolants computed for the global refinement, the four error paths that end in the four target states, as well as the relevant abstract states. The tree structure should be interpreted like the following. After going from N4 to N6, the interpolant `{[a = 1]}` at N6 indicates that the valuation of **int** a is needed for excluding an infeasible error path — in this case the infeasible error path leading to the **assert** statement in line 7 is excluded. Coming from N6 and going on to N10, no information is needed for excluding any infeasible error path, as indicated by the “empty” interpolant at N10. But right after that, the interpolant `{[b = 1]}` is strong enough to exclude the remaining infeasible error paths. With global refinement enabled, the refinement component has to compute this tree of interpolants instead of a single sequence of interpolants. Of course it would be possible to extract each error path from this tree and compute interpolants for each path individually, but this would waste two opportunities for optimizations. First, it is possible to skip the interpolation of some error paths completely, because some

```
 1 #include <assert.h>
 2 int main() {
 3
 4   int a = 1;
 5
 6   if (a == 0) {
 7     assert(0);
 8   }
 9
10   int b = 1;
11
12   if (b == 0) {
13     assert(0);
14   }
15
16   else if (b == 2) {
17
18     int c = 1;
19
20     if (c == 0) {
21       assert(0);
22     }
23
24     else if (c == 2) {
25       assert(0);
26     }
27   }
28 }
```

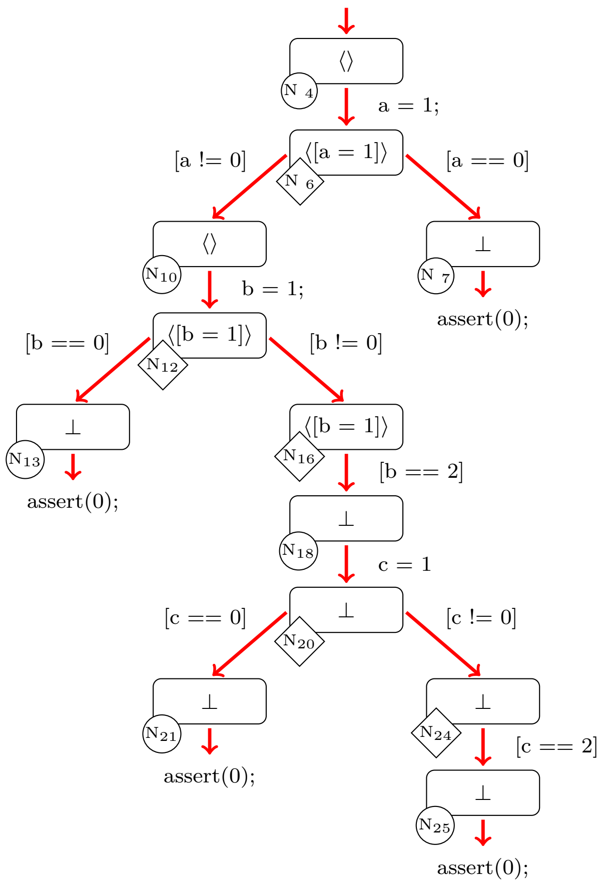


Figure 5.4: A simple verification task with an interpolant tree

target states might already be excluded by interpolants computed for a target state for which the interpolation was performed earlier. Second, redundant interpolations over common prefixes of infeasible error paths can sometimes be avoided, because some error paths share a common prefix with other error paths. In the example here we can avoid the interpolation of the error path to the **assert** statement in line 25, because the interpolant at the branching state N20 is already set to $\perp$, meaning that an interpolant further up the tree already guarantees that this state is unreachable. Moreover, assume that the path to the **assert** statement in line 13 was already interpolated over, and we decide to interpolate next over the path to the **assert** statement in line 21. Here we can check, if the already existing interpolant `{[b = 1]}` at the branching state N12 is strong enough to serve as initial interpolant for the partial interpolation of the suffix from N12 to the **assert** statement in line 21. As that is the case, we can save some interpolation queries, and in addition, we can assure that the precision stays smaller, because no new program variables are added to the precision. To sum up, in this example one single refinement is enough to exclude all infeasible error paths, while an approach using the normal CEGAR approach might need up to four individual refinements.

So, in theory, the reduction of refinements as well as potential for optimizations in the interpolation procedure are what makes global refinement attractive. However, when comparing this to the results from Section 5.4, of course under the same experimental setup, it turns out that global refinement does not lead to any significant improvement, because for the total of 4 283 verification tasks it only solves 4 verification tasks more than the default refinement procedure is able to solve.

We conclude from this that the effort saved through the improvements for the interpolation and refinement procedures are roughly the same as the overhead introduced with continuing the state-space exploration after having identified the first target state. This result is not a total surprise, because with the improvements described earlier in this chapter, the limiting factor of the value analysis with CEGAR seems to be the construction of the abstract model. Sometimes this model is concise, because suitable interpolants are identified, but sometimes it is not, because the interpolants that are identified lead, for example, to repeated loop unrollings. Whether or not using global refinement has no *direct* influence on the choice of interpolants, so only by chance one approach may lead to more suitable interpolants and a more concise abstract model, and only in those few cases one of the approaches is significantly faster than the other. Therefore, we do not investigate the concept of global refinements any further, but we refer to the technique presented in Chapter 7, because it enables us to guide the interpolation process towards suitable interpolants, which then allow the analysis to obtain a more concise abstract model which leads to a more efficient verification process.

5.6.3 Impact-Like Refinement for the Value Analysis

Finally, we also experimented to combine the value analysis with lazy abstraction with interpolants [88], a technique which was first introduced for the predicate domain. In the original work it was shown that the underlying algorithm, called Impact, was far more efficient than the approach taken by Blast [18], which by that time was the best implementation of an analysis based on predicate abstraction. The reason why the Impact approach may perform better for a given verification task is due to the fact, that it may solve verification tasks with far less refinements than classic lazy predicate abstraction. More recent work [33] formalized both approaches in an unified algorithm, which can be parametrized to either perform lazy predicate abstraction (Blast) or lazy abstraction with interpolants (Impact). This work shows that adding adjustable-block encoding (ABE) [24] to this unified algorithm, and thus making it available to both approaches, closes the gap between lazy predicate abstraction (Blast) and lazy abstraction with interpolants (Impact).

For the value analysis, there is no concept available like ABE, and therefore, again with the goal of finding counterexamples faster with the value analysis, we propose a refinement schema for the value analysis that is based on global refinement (cf. Section 5.6.2) and also borrows ideas from the Impact algorithm.

For detailed insights on the Impact algorithm we refer to the literature [33, 88], and we only focus here on the most relevant bits. The main difference between lazy predicate abstraction (Blast) and lazy abstraction with interpolants (Impact) is that after a refinement the later does not delete and re-explore states in the ARG, but instead it strengthens the abstract states by conjunctively adding the corresponding interpolant directly to the state formula wrapped in the abstract state. This strengthening effects the coverage relation between abstract states, and thus might lead to inconsistencies in the coverage relation. The algorithm has to restore the coverage relation to a consistent state, which comes at an extra cost.

We adapt the idea of the Impact algorithm to strengthen abstract states of the value analysis with interpolants provided by the value-analysis interpolation (cf. Section 4.6). If the value analysis finds a target state, interpolants are computed for the respective error path, and similar to the Impact algorithm, we strengthen the abstract states, i. e., the abstract variable assignments, with the value-domain interpolants, which can also be interpreted as abstract variable assignments. In contrast to Impact, we deliberately leave the coverage relation inconsistent after strengthening abstract states in a refinement in order to save the extra cost.

For our intent this is acceptable, because our goal is to find counterexamples more rapidly. However, we need to deal with a few consequences. Assume we find a target state with this approach. If it turns out to be a real counterexample, the algorithm terminates and returns the verdict `false`. If the error path is infeasible a refinement is performed, in which the abstract states are strengthened. This possibly leaves the coverage relation in an inconsistent state. Despite that, the algorithm continues the exploration, and it either finds another target state and performs again a refinement by strengthening the abstract states using interpolants, or no more target states are found. In the later case, we cannot return the verdict `true` for the verification task, because the current ARG represents an *under-approximation*, due to a possibly inconsistent coverage relation. So once the state-space exploration is complete, our approach performs a full restart of the state-space exploration in order to rebuild the ARG with a consistent coverage relation. Only if the ARG resulting from such a re-exploration is free of target states, then we can return the verdict `true` for the verification task. If another target state is found, then this refinement loop starts anew.

This approach indeed finds some counterexamples faster than the CEGAR approach we presented in earlier chapters. Despite that, we did not pursue this concept any further, because in our evaluation —performed in the same experimental setup as before— it does not only perform worse in proving `true` verdicts, which was expected, but also is not competitive in regard to finding counterexamples. Especially on the verification tasks in the categories `ECA` and `ProductLines` this approach fails to find counterexamples efficiently. As already stated earlier (cf. Section 5.4), a good strategy for these verification tasks is to track program variables globally right from the start, but this contradicts the Impact approach, which refines a single infeasible error path after the other, and, instead of continuing the re-exploration after a refinement at the initial state, it continues deep in the ARG, similar to the value analysis based on CEGAR and lazy abstraction (cf. Section 4.8), which also did not work well for the categories `ECA` and `ProductLines`. We could improve this for the Impact approach by restarting the analysis more frequently, i. e., be more eager, but then this would be almost identical to the standard value analysis with CEGAR, leaving us without any justification for having the Impact-like approach in the first place.

5.7 Conclusion

In this chapter we presented several techniques that allow the value analysis with CEGAR to become a competitive analysis.

The most important step that we needed to take was to lower the number of refinements that the original CEGAR component of the value analysis requires. We achieved that goal by applying a scoped precision and by restarting the analysis after each refinement. Several optimizations for the interpolation procedure further speed up the analysis.

In addition, we detailed on the versatility of the refinement component of the value analysis, and we reported on several ideas that, despite not living up to our expectations, at least firmed our belief that the key for optimizing the CEGAR component lies in finding better interpolants.

5.7.1 Lessons Learned

An interesting insight is that lazy abstraction, for the benchmark set we use, is not beneficial for the value analysis. A location-based precision, i. e., a parsimonious precision, is one of the two main arguments brought forward by the lazy-abstraction principle to facilitate an efficient verification process. However, for the value analysis, this reasoning does not apply, because the same information is needed in many different paths, making a location-based precision too fine-grained and leading to many repeated refinements. If deciding against lazy abstraction, then the value analysis with CEGAR clearly out-performs the plain value analysis in terms of verification efficiency, becoming a competitive analysis on its own.

5.7.2 Challenge

With the findings from this chapter we have available a competitive analysis, which is able to provide a verdict for many verification tasks from our benchmark set. However, the addition of CEGAR to the value analysis does not make the analysis more precise, and as such, the analysis still returns an incorrect `false` verdict for some verification tasks, especially for verification tasks where the reachability of the target state depends on non-determinism, e. g., caused by uninitialized variables. So before further improving the verification efficiency of the value analysis, we now turn the attention to the design of an efficient and *precise* analysis.

5.7.3 Proposition

A straight-forward approach would be to add symbolic capabilities directly to the value analysis. We did not follow this path, because we would risk breaking one of the main design decisions of the value analysis, namely its simple, low-overhead approach, which so far has proven highly valuable. So instead of tightly integrating symbolic capabilities to the value analysis, we rather make use of what the CPAchecker framework provides, which already has highly capable symbolic analyses on board.

5.7.4 Solution

In order to obtain a precise and efficient analysis based on the current state of the value analysis, we suggest to combine the value analysis with the existing predicate analysis of CPAchecker, with both analyses making use of the CEGAR approach. In order to allow this novel composite analysis to stay as efficient as possible, the value analysis will remain to be the main driver of the composite analysis, and it will only turn to the predicate analysis when encountering an infeasible counterexample that it cannot refute itself. The next chapter will cover the details of this novel approach.

6 Precise and Efficient Composite Analysis based on CEGAR

The optimizations detailed in the previous chapter are targeted at the verification effectiveness and verification efficiency of the value analysis, suggesting several improvements for the refinement approach and the interpolation procedure in the value domain. In this section we continue to improve the verification effectiveness of the analysis, but now by lowering the number of incorrect `false` verdicts that the value analysis reports, while still guaranteeing an efficient verification process.

6.1 Motivation

Reporting incorrect `false` verdicts greatly hinders the verification effectiveness of any analysis, and so far in this work we neglected to limit the number of incorrect `false` verdicts which the value analysis reports. One reason why the value analysis raises false alarms is due to the aforementioned non-determinism that, at least to some degree, most verification tasks expose. This non-determinism is introduced on purpose by the authors of the verification task, e. g., to simulate user input or communication with external libraries, with the goal to generalize the verification task such that the size of the state space that the analysis has to cover is increased. Without any non-determinism, each verification task would represent a path program, and the verification of these is not of major interest.

```c
#include <assert.h>
int main() {

  int a;

  if (a != 1) {
    if (a == 1)
      assert(0);
  }
}
```

Figure 6.1: Example verification task exposing non-determinism due to missing initializer

By design, the value analysis is not particularly good in dealing with non-determinism, making it prone to reporting false alarms, as shown by the tiny example verification task in Figure 6.1. This verification task defines a local variable **int** a, that does not have an initializer. Therefore, a verifier has to explore both the then branch as well as the **else** branch of the assume operation [a != 1]. The value analysis is not able to derive any valuation for the variable **int** a from the assume operation `[a != 1]` in the `then` branch, and eventually will also compute a successor for the `then` branch of the assume operation `[a == 1]`, despite the fact that the two assume operations `[a != 1]` and `[a == 1]` obviously are contradicting. A symbolic analysis, like an analysis based on predicate abstraction [60] or symbolic execution [78] can easily pick up this contradiction and prove the verdict `true` of the verification task, but the value analysis for itself reports a false alarm instead.

Again, we clearly distance ourselves from introducing any symbolic capabilities into the value analysis, because we believe that the straightforwardness of the value analysis is one of its major advantages. Instead, to resolve this dilemma and to maintain the straightforwardness of the value analysis while also allowing an efficient and effective verification process, we suggest to combine the value analysis with a predicate analysis [24].

6.2 Related Work

Predicated lattices [57] is a concept that joins data-flow analyses with a predicate analysis. The idea is to increase the precision of resulting data-flow analysis, as not only lattice elements are tracked but also a set of predicates. The latter partitions the state space of the verification task in such a way that the normally path-insensitive data-flow analysis becomes as path-sensitive as needed, i. e., paths that must be analyzed in isolation are not joined.

Dynamic precision adjustment [22] is another concept that also combines a value analysis and a predicate analysis. There, the verification effort imposed on either of the analyses depends on a predefined, static threshold. Initially, a variable is tracked by the value analysis, but once the number of distinct valuations of this variable exceeds the threshold, the variable is tracked symbolically by the predicate analysis.

In contrast to these two approaches our composite analysis performs CEGAR in both the value analysis and the predicate analysis, and therefore, because only relevant facts are tracked by either one of the component analyses, each can be run in a path-sensitive configuration and still avoid the problem of state-space explosion in many cases.

Just recently, a CEGAR framework targeted at symbolic transition systems was presented [63]. This approach also combines an explicit analysis and a predicate analysis, with both analyses incorporating CEGAR, interpolation and lazy abstraction, making it quite similar to our approach. While the authors state that their prototype implementation is unable to compete with state-of-the-art tools, they confirm our results that the composite analysis leads to a performance boost compared to running the explicit analysis or the predicate analysis in isolation.

Besides performing CEGAR in both component analyses, another novelty of our approach is that the decision which information is tracked by which analysis is based on the level of expressiveness of the component analyses. The details of our approach are subject of the next section.

6.3 Composition of a Value Analysis and a Predicate Analysis

The analysis, as composition of the value analysis and the predicate analysis, works basically the same as any analysis based on CEGAR (cf. Algorithm 2). The analysis is started with an empty precision, i. e., neither the value analysis nor the predicate analysis track any variables or predicates, respectively. If this composite analysis finds an error path, then this path is first checked for feasibility in the value domain. If it is infeasible, then a refinement of the value domain is performed (cf. Section 4.7), and the composite analysis continues the CEGAR loop. If the path is feasible according to the semantics of the value domain, then the path is also checked for feasibility in the predicate domain. If the predicate domain confirms the feasibility of the path, the verdict `false` is returned with this error path as counterexample.

If, however, the path is found to be infeasible under the semantics of the predicate domain, then the value domain is not expressive enough to refute that program path, e. g., due to non-determinism encoded in the verification task. In such a case, we ask the predicate analysis to refine its precision along that path, which yields a refined predicate precision that eliminates this error path by considering facts along that path in the predicate domain.

We argue that, in general, the post-operations of the predicate analysis are more expensive than the post-operations of the value analysis, hence, we always try to perform a refinement for the, supposedly, cheaper value analysis first, and only use refinements for the predicate analysis as fallback, in case the expressiveness of the value analysis does not suffice to exclude an infeasible error path.

As a further enhancement, and similar as for dynamic precision adjustment [22], the composite analysis can decide to remove variables from the precision in the value domain once the number of different valuations for a variable along a path exceeds a certain threshold. Because these variables are relevant for excluding an infeasible error path —after all they were part of the value analysis precision— a later predicate refinement will automatically add predicates about these variables to the precision in the predicate domain. We regard this as a particularly nice property of our approach, because when we decide to manually decrease the precision of the value analysis, then the auxiliary predicate analysis will *automatically* compensate the loss of precision of the overall analysis.

In conclusion, after a refinement step, either the precision of the value analysis is refined, which normally is preferred, or the precision of the predicate analysis is refined. Note that this refinement-based, parallel composition of a value analysis and predicate analysis is strictly more powerful than a mere parallel product of the two analyses, because the value domain tracks exactly what it can efficiently analyze, while the predicate domain takes care of what is beyond that, resulting in an analysis that allows both an efficient and effective verification process, as shown in the evaluation in the next section.

6.4 Evaluation

In the following we present the results of evaluating the composition of the value analysis and the auxiliary predicate analysis described above. The main purpose of this evaluation is to show that the addition of the auxiliary predicate analysis to the value analysis (1) still allows for an efficient verification process, i. e., despite the now increased precision the composite analysis still remains efficient, and that the addition of the auxiliary predicate analysis to the value analysis (2) reduces the number of incorrect `false` verdicts which the stand-alone value analysis reports.

Again, in order to be able to draw comparisons with the evaluations from the previous Sections 3.5, 4.8 and 5.4, we reuse the same benchmark verification tasks in the identical experimental setup, and we again take CPAchecker in revision 20 406. The benchmarking framework BenchExec is used and configured the same way as before.

6.4.1 Configuration

The composite analysis is based on the configuration of the value analysis with improved CEGAR from Section 5.4, but is extended by a CPA performing predicate analysis, which is configured to perform abstraction computations at loop-head locations and locations where control flow joins [24]. The predicate analysis relies on SmtInterpol [40] as SMT solver and interpolation engine. In addition, a custom refiner is put in place that, according to the logic described above, either delegates to the refiner for the value analysis or to the refiner for the predicate analysis. As in all previous evaluations we again disable explicit recursion and counterexample checks.

In order to allow reproducibility of the evaluation, an example for a complete command line to run this composite analysis based on CEGAR as well as the full results and raw data are available on our supplementary web page [1].

[1] `http://www.sosy-lab.org/research/phd/loewe/#CompositeAnalysisBasedOnCEGAR`

	BitVectorsReach	ControlFlow	DeviceDriversLinux64	ECA	Floats	Loops	ProductLines	Sequentialized	Simple	Overall
total	48	48	2 120	1 140	81	141	597	62	46	4 283
unsolved	3	3	509	605	7	49	149	23	0	1 348
solved	45	45	1 611	535	74	92	448	39	46	2 935
correct	26	45	1 562	535	42	85	448	39	46	2 828
true	20	27	1 395	264	23	47	279	16	22	2 093
false	6	18	167	271	19	38	169	23	24	735
incorrect	19	0	49	0	32	7	0	0	0	107
true	5	0	2	0	2	0	0	0	0	9
false	14	0	47	0	30	7	0	0	0	98

Table 6.1: Table showing the verification effectiveness of the composition of the value and predicate analysis

6.4.2 Results

For the sake of completeness, we show an overview of the verification effectiveness of the composition of the value analysis and the predicate analysis in Table 6.1. The most important fact we can draw from this table is that the composite analysis indeed allows for an efficient verification process, as it solves 69 % of the 4 283 verification tasks, so we can rightfully claim that hypothesis 1 from above is fulfilled.

In order to get a better impression of the differences in verification effectiveness of the composition of the value analysis and the predicate analysis and the stand-alone value analysis with improved CEGAR we show their differences in verification effectiveness first via Table 6.2, which represents the cell-wise difference of Table 6.1 and Table 5.2, respectively, i. e., similar as already done before in Sections 4.8.3 and 5.4.3.

Note that there is no category where the number of solved instances is increased for the composition of the value analysis and the predicate analysis, and in total the more precise approach solves 153 tasks less.

There are two main reasons for this, which we discuss shortly. First, because of the addition of the auxiliary predicate analysis, more effort is spent to allow a more precise analysis. This extra effort is even spent when it would actually not be needed. For example, in the categories ECA and ProductLines no predicate refinements are needed (cf. row refinement$_{\mathrm{PA}}$), but there alone 64 verification tasks less can be solved

	BitVectorsReach	ControlFlow	DeviceDriversLinux64	ECA	Floats	Loops	ProductLines	Sequentialized	Simple	Overall
total	48	48	2 120	1 140	81	141	597	62	46	4 283
solved$_{\Delta}$	−1	0	−57	−52	−6	−22	−12	−3	0	−153
correct$_{\Delta}$	1	0	−21	−52	5	22	−12	8	19	−30
true$_{\Delta}$	6	0	5	−23	8	25	−4	8	19	44
false$_{\Delta}$	−5	0	−26	−29	−3	−3	−8	0	0	−74
incorrect$_{\Delta}$	−2	0	−36	0	−11	−44	0	−11	−19	−123
true$_{\Delta}$	5	0	2	0	2	0	0	0	0	9
false$_{\Delta}$	−7	0	−38	0	−13	−44	0	−11	−19	−132
refinement$_{\text{PA}}$	23	0	177	0	32	61	0	11	38	342
false$^{\text{incorrect}}_{\text{VA}} \mapsto$ true$^{\text{correct}}_{\text{PA}}$	6	0	18	0	8	25	0	8	19	84
false$^{\text{incorrect}}_{\text{VA}} \mapsto$ unsolved$_{\text{PA}}$	1	0	20	0	5	19	0	3	0	48
false$^{\text{incorrect}}_{\text{VA}} \mapsto$ false$^{\text{incorrect}}_{\text{PA}}$	14	0	47	0	30	7	0	0	0	98
false$^{\text{correct}}_{\text{VA}}$ & refinement$_{\text{PA}}$	2	0	97	0	0	14	0	0	19	132

Table 6.2: Table showing the verification effectiveness of the composition of the value and predicate analysis in comparison to the value analysis with improved CEGAR

with the compositional approach, because the predicate analysis performs SAT checks as the analysis progresses[2]. We already discussed that the verification tasks in the categories ECA or ProductLines have a rather deterministic nature, which allows the stand-alone value analysis to solve these verification tasks efficiently, while the effort of the compositional approach basically gets wasted here. But of course, not all of the effort of the compositional approach is wasted, after all, with it 44 more correct `true` verdicts are reported. This is mainly because the predicate analysis can deal better with the loop structures occurring in the categories BitVectorsReach and Loops, and also has support for pointer-aliasing that occurs in the category Simple, allowing the compositional approach to outperform the stand-alone value analysis.

The second reason why the compositional approach solves less verification tasks in total is because it avoids a lot of incorrect `false` verdicts that the stand-alone

[2]As possible optimization, the implementation could delay SAT checks to after the first unsuccessful value-analysis refinement, i. e., to the point where it is clear that the value analysis alone is not capable of analyzing the verification task alone, but this would introduce coupling between the two CPAs, so this optimization is not implemented.

value analysis reports, i. e., hypothesis 2 from above is fulfilled as well. From the 230 incorrect `false` verdicts the stand-alone value analysis reports, the compositional approach avoids 132 of those, as it turns 84 into correct `true` verdicts, 48 run out of resources with the compositional approach, while 98 remain incorrect `false` verdicts. The latter case may happen because the SAT-based predicate abstraction and its implementation as predicate analysis within CPAchecker are not able to reason about each and every detail of the programming language C, so incorrect answers remain an issue.

Also, note row $\text{false}_{\text{VA}}^{\text{correct}}$ & $\text{refinement}_{\text{PA}}$ from Table 6.2. This row shows the sum of verification tasks where the stand-alone value analysis reported a supposedly correct `false` verdict, however, when combined with the predicate analysis, the respective counterexample is refuted by the predicate analysis in a total of 132 cases. So the addition of the predicate analysis does not only allow to avoid incorrect `false` verdicts, it also significantly raises the confidence that a `false` verdict in fact represents a bug in the respective verification task. Note that we do not claim to perform witness checking here [16] but rather propose an approach to integrated counterexample checking.

6.5 International Competition on Software Verification 2013

The combination of the value analysis and the predicate analysis demonstrated its potential in SV-COMP'13[3], winning the silver medal in the categories ControlFlow, DeviceDriversLinux64, SystemC, as well as in Overall [11, 83]. That edition of SV-COMP also introduced the notion of a score-based quantile plot, which are particularly helpful for visualizing the different aspects of verification quality [11]. In such a score-based quantile plot each data point `(x, y)` of a graph yields the maximum run time `y` for the `n` fastest correct verification results with the accumulated score `x` of all incorrect results and those `n` correct results [11]. Hence, the graph of a verification tool that reports many incorrect verdicts starts far in the negative range, while a precise verification tool will start around an accumulated score of 0. If the graph corresponding to a verification tool spreads over large parts of the x-axis, then this means that the verification tool returns many correct answers, and the verification tool whose graph stretches the farthest to the right is regarded as the most successful, as it achieves the highest score.

In Figure 6.2 we show the score-based quantile plot comparing the stand-alone value analysis with improved CEGAR, the composition of the value analysis and the predicate analysis, and the default predicate analysis of CPAchecker [24, 86]. From the graphs we can easily identify some key differences between the approaches.

[3]`http://sv-comp.sosy-lab.org/2013/results/`

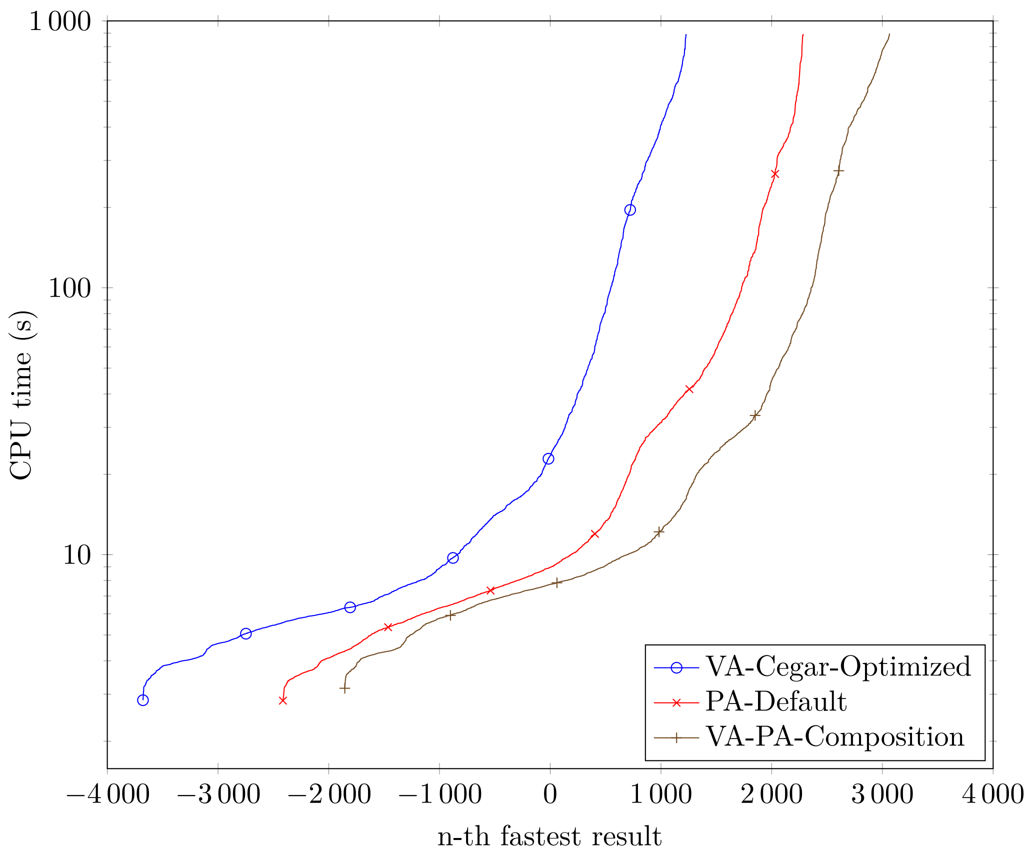


Figure 6.2: The score-based quantile plot, based on the scoring schema of SV-COMP'16, for the value analysis with improved CEGAR, the default predicate analysis of CPAchecker, and the composition of the two analyses

First, the graph for the stand-alone value analysis with improved CEGAR starts furthest to the left, i. e., in the more negative range of the x-axis, which signals that the stand-alone value analysis with improved CEGAR reports more incorrect verdicts. Second, the graph for the composition of this value analysis and the predicate analysis stretches further to the right, i. e., in the more positive range of the x-axis, which signals that this approach has a higher overall verification quality, because it reports less incorrect verdicts, and at the same time, also performs about the same amount of successful, correct verification work as the stand-alone value analysis with improved CEGAR. The graph of the default predicate analysis runs between the two other graphs, i. e., the predicate analysis performs somewhere between the two other approaches. In total, the stand-alone value analysis obtains a score of 1 227, while the composition of this value analysis and the predicate analysis achieves a total of 3 065 points, and comes out of the comparison as the clear winner.

6.6 Conclusion

This section introduced a novel composition of a value analysis and a predicate analysis, where the verification effort is split up based on the expressiveness of the two domains. Letting the predicate analysis accompany the value analysis allows for a more precise (composite) analysis that still is efficient, and which is on a par with the world's leading tools for software verification, as attested by SV-COMP'13.

6.6.1 Lessons Learned

We achieved a significant improvement, especially in terms of verification effectiveness, by building a composite analysis of the value analysis and a predicate analysis that integrates a CEGAR approach over *both* analyses. Because both analyses complement each other very well, the composite analysis forms a verification approach that is more effective and efficient than both the value analysis or the predicate analysis running on their own.

6.6.2 Challenge

Of course we strive to design an analysis that can solve as many verification tasks as possible. In its current form, the value analysis with CEGAR performs well on many benchmarks, but still suffers from state-space explosion occasionally. Combining the value analysis and the predicate analysis avoids many incorrect `false` verdicts, but same as for the value analysis, the predicate analysis also diverges for many verification tasks, because the analyses are not always able to find suitable abstractions for a given verification task.

6.6.3 Proposition

In order to avoid divergence of both the value and the predicate analysis we propose to improve the *quality* of the abstract model. The size and form of the abstract model is influenced by the precision that the analysis employs, and, as both analyses are based on CEGAR with interpolation, the precision is build from interpolants. So in order to improve the quality of the abstract model, we actually need to improve the quality of the interpolants.

6.6.4 Solution

Often times, an infeasible error path contains several reasons of infeasibility, and in theory, one could compute interpolants, extract a precision and perform a refinement based on any of these reasons of infeasibility. However, both for the value and the predicate analysis, the standard interpolation engines do not allow any control over the interpolation process. Therefore, we propose an algorithm that allows to extract from one single infeasible error path a set of infeasible paths, where each one can be used for precision refinement, thus enabling the possibility of selecting interpolants such that different abstract models of different quality can be computed.

Also note that, now, with the value analysis being fit for CEGAR, any advances regarding CEGAR or interpolation can be applied to the predicate analysis as well as to the value analysis, or any analysis that implements refinement based on value interpolation (cf. Section 4.7), e. g., the analyses based on the octagon domain or on symbolic execution, both which are available in CPAchecker (cf. Section 5.5).

7 Refinements over Infeasible Sliced Prefixes

In Chapter 4 we defined CEGAR for the value analysis, and in Chapter 5 several optimization to the CEGAR approach were presented, that allow the value analysis with CEGAR to be competitive on a wide range of verification tasks. Additionally, in Section 5.6 we elaborated on several techniques to further increase the performance of the value analysis with CEGAR, which, however, fell short of our expectations, such that there are still verification tasks that the plain value analysis can solve but the value analysis with CEGAR cannot solve. This indicates that the abstract model being built during the CEGAR iterations is not ideal, and in the following we present a technique that may help an analysis based on CEGAR to steer away from the problem of state-space explosion in many cases.

7.1 Motivation

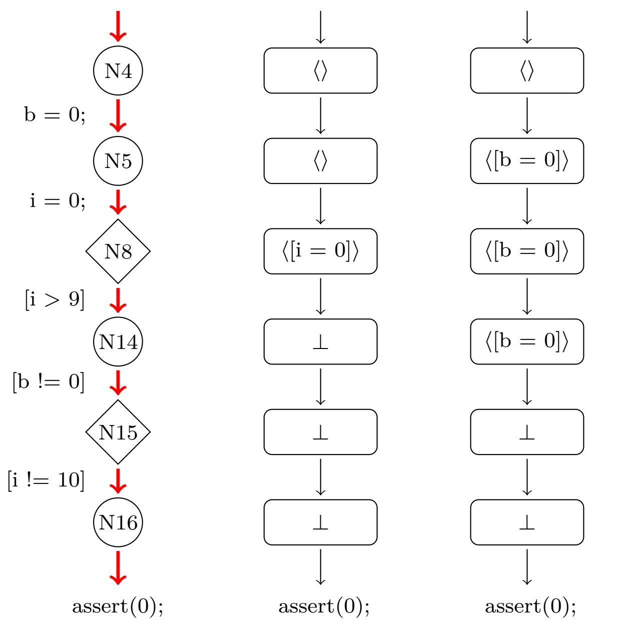
There are a couple of explanations for the effect described above. Specifically for the value analysis, one explanation is that several verification tasks expose only very little non-determinism or even are fully deterministic. As already motivated in Section 5.4, the full precision of the plain value analysis does no harm there, because, in the extreme, only a single program path has to be explored, while the value analysis with CEGAR first has to find a suitable abstraction by learning fact after fact from successive refinements.

Another reason that can explain the divergence of an analysis based on CEGAR lies in the fact that there exists not only a single abstraction for a given verification task, but in general infinitely many —some of which are more suitable than others— and it can easily happen that the analysis diverges because no suitable abstraction is found for a given verification task.

Ultimately, in the context of analyses using interpolation-based CEGAR, the interpolants dictate how the abstract model of a verification task will finally look like, and therefore the choice of interpolants influences the performance of the analysis significantly. Figure 7.1 gives an example. In this verification task, the analysis will typically find the shown error path, which is infeasible for two different reasons. Both the value of `int i` and the value of `int b` can be used to find a contradiction

```
#include <assert.h>
extern int f(int x);
int main() {
  int b = 0;
  int i = 0;

  while(1) {
    if(i > 9) {
      break;
    }
    f(i++);
  }

  if(b != 0) {
    if(i != 10) {
      assert(0);
    }
  }
}
```

(a) verification task



(b) error path (c) bad sequence (d) good sequence

Figure 7.1: From left to right, (a) a verification task, (b) an infeasible error path, and a (c) "bad" interpolant sequence and a (d) "good" interpolant sequence for this infeasible error path

in the error path. In general, it is now beneficial for the verifier to track the value of the variable `int b` —having boolean character— and not to track the value of the variable `int i` —being a loop-counter variable— because the latter has many more possible values, and tracking it would usually lead to an expensive unrolling of the loop. Instead, if only variable `int b` is tracked, the verifier can conclude the verdict `true` of the verification task without unrolling the loop. Thus, we would like to use for refinement the interpolant sequence shown on the right, with only the boolean variable, and not the left one with the loop-counter variable. However, interpolation engines typically do not allow to guide the interpolation process towards "good", or away from "bad" interpolant sequences. Mind that interpolation engines inherently cannot do a better job here. They do not have access to information such as whether a specific variable is a loop counter and should be avoided in the interpolant. Instead, which interpolant is returned depends solely on the internal algorithms of the interpolation engine. For the illustrative example above procedure Interpolate (cf. Algorithm 3) would return the "bad" interpolation sequence to the left, and the analysis cannot avoid this because the interpolation engine does not allow any control over the interpolation engine from the outside. The same applies to most SMT-based model checkers, which often rely on off-the-shelf interpolation engines. Normally these cannot be controlled on such a fine-grained level, and the model checker querying the interpolation engine is stuck to what the interpolation engine returns, be it good or bad for the verification process.

The straight-forward approach to allow more control over the interpolation process is to implement such a feature into the interpolation engine of choice, however, as of now, CPAchecker has implementations for two interpolation algorithms —one for the value analysis and one for analysis based on symbolic execution— and maintains interfaces to four different SMT-based interpolating solvers to be used together with the predicate analysis. Implementing such a feature in a single interpolation engine is already a considerable effort, so implementing and maintaining this for all supported solvers and interpolation algorithms would be tedious. Instead, we propose a technique that is capable of remodeling a given interpolation problem in such a way, that we can extract different interpolants from it, thus enabling a selection process to be incorporated into any interpolation-based CEGAR algorithm. We expect that, in many cases, this allows a successful verification process where the standard approach diverges. Furthermore, this approach is highly versatile, because it is independent from both specific properties of the interpolation engines and from particular characteristics of the abstract domains, allowing its application in SMT-based predicate analysis, as well as in analyses based on numeric abstract domains, like for example the value analysis or the octagon analysis.

Finally, note that, while we use interpolation to compute the refined precisions, our method is not bound to interpolation, because invariant-generation techniques for refinement, such as path invariants [20], can equally benefit from the new possibility of selection.

7.2 Related Work

The desire to control which interpolants an interpolation engine produces, and trying to make the verification process more efficient by finding good interpolants, is not new. The first work in this direction suggested to control the *interpolant strength* [52] such that the user can choose between strong and weak interpolants. This approach is unfortunately not implemented in standard interpolation engines, and it requires to rewrite the algorithm that extracts interpolants from resolution proofs.

The technique of interpolation abstractions [96], a generalization of term abstraction [5], can be used to guide solvers to pick good interpolants. This is achieved by extending the concrete interpolation problem by so called templates (e. g., terms, formulas, uninterpreted functions with free variables) to obtain a more abstract interpolation problem. An interpolant for the abstract interpolation problem is also a solution to the concrete interpolation problem. Suitable interpolants can be chosen using a cost function, because these interpolation abstractions form a lattice. In contrast to interpolation abstractions, our approach does not rely on SMT solving and is independent from the interpolation engine and abstract domain, so it is also applicable to, e. g., the value and the octagon domain.

Path slicing [77] is a technique that was introduced to reduce the burden of the interpolation engine. Before the constraints of the path are given to the interpolation engine, the constraints are weakened by removing facts that are not important for the infeasibility of the error path, i. e., a more abstract error path is constructed. In our approach, we also make the error path more abstract, but in different directions to obtain different interpolant sequences, from which we can choose one that yields a suitable abstract model. While path slicing is interested in reducing the run time of the *interpolation engine* by omitting some facts, we are interested in reducing the run time of the *verification engine* by spending more time on interpolation and selection allowing us to create a better abstract model.

SMT solvers can extract unsatisfiability cores [42] from a proof of unsatisfiability, and there is an analogy between a set of unsatisfiability cores extracted from a formula and the approach proposed here. However, our approach is more general, because it is applicable also to domains that are not based on SMT formulas, such as value domains. Furthermore, SMT solvers typically strive for small unsatisfiability cores [42], but this alone does not guarantee a verifier to be effective. It would be interesting to investigate the extraction of several unsatisfiability cores during a single refinement, with the goal of selecting a refinement based on characteristics of the unsatisfiability core, similar as proposed by our work here.

The software verification framework Ultimate Automizer computes a proof of infeasibility for a trace in form of a single inductive sequence of state predicates, e. g., via Craig interpolation. Current versions of Ultimate Automizer compute not only a single inductive sequence, but two, one via the strongest–post-condition predicate transformer, and one via the weakest-precondition predicate transformer. This improves the generalization from the infeasible trace to a set of infeasible traces, and thus more infeasible traces can be regarded as irrelevant in the further course of the analysis [67]. By design, this approach is limited to solving at most two different interpolation problems, while our technique usually leads to a wide choice of different interpolation problems during a single refinement step, as shown in the evaluation section later in this chapter.

7.3 Introducing Infeasible Sliced Prefixes

An analysis based on CEGAR encounters an infeasible error path if the precision is too coarse. An infeasible error path contains at least one assume operation for which the reachability algorithm computes a non-contradicting *abstract* successor based on the current precision, but computes a contradicting successor if the *concrete* semantics is used. Every infeasible error path contains at least one such contradicting assume operation, but often there exist several independently contradicting assume operations in an infeasible error path, which leads to the notion of sliced prefixes.

A path $\varphi = \langle (op_1, l_1), \ldots, (op_w, l_w) \rangle$ is a *sliced prefix* for a program path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ if $w \leq n$ and for all $1 \leq i \leq w$, we have $\varphi.l_i = \sigma.l_i$ and ($\varphi.op_i = \sigma.op_i$ or ($\varphi.op_i = [true]$ and $\sigma.op_i$ is assume op)), i. e., a sliced prefix results from a program path by omitting pairs of operations and locations from the end, and possibly replacing some assume operations by no-op operations. If a sliced prefix for σ is infeasible, then σ is infeasible.

7.4 Extracting Infeasible Sliced Prefixes

Algorithm 6 extracts from an infeasible error path a set of *infeasible* sliced prefixes. The algorithm iterates through the given infeasible error path σ. It keeps incrementing a *feasible* sliced prefix σ_f that contains all operations from σ that were seen so far, except contradicting assume operations, which were replaced by no-op operations. Thus, σ_f is always feasible. For every element (op, l) from the original path σ (iterating in order from the first to the last pair), it is checked whether (op, l) contradicts σ_f, which is the case if the result of the strongest post-operator for the path $\sigma_f \wedge (op, l)$ is contradicting (denoted by $\perp$). If so, the algorithm has found a new infeasible sliced prefix, which is collected in the set Σ of infeasible sliced prefixes.

Algorithm 6: ExtractSlicedPrefixes(σ)

```
Input     : an infeasible path $\sigma = \langle (op_1, l_1), \ldots, (op_m, l_m) \rangle$
Output    : a non-empty set $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ of infeasible sliced prefixes of $\sigma$
Variables : a path $\sigma_f$ that is always feasible

1 $\Sigma := \emptyset$
2 $\sigma_f := \langle\rangle$
3 foreach $(op, l) \in \sigma$ do
4     if $\widehat{\mathsf{SP}}_{\sigma_f \wedge (op,l)}(\top) = \perp$ then
          // add $\sigma_f \wedge (op, l)$ to set $\Sigma$ of infeasible sliced prefixes
5         $\Sigma := \Sigma \cup \{\sigma_f \wedge (op, l)\}$
          // append no-op
6         $\sigma_f := \sigma_f \wedge ([true], l)$
7     else
          // append original pair
8         $\sigma_f := \sigma_f \wedge (op, l)$
9 return $\Sigma$
```

The feasible sliced prefix σ_f is extended either by a no-op operation (Line 6) or by the current operation (Line 8). When the algorithm terminates, which is guaranteed because σ is finite, the set Σ contains infeasible sliced prefixes of σ, one for each reason of infeasibility. There is always at least one infeasible sliced prefix because σ is infeasible.

Mind that Algorithm 6 only needs an operator for computing abstract successors, e. g., $\widehat{\mathsf{SP}}$, for which contradiction must be defined, i. e., there must be means to check if $\widehat{\mathsf{SP}}_\sigma(\top) = \bot$ holds for a path σ. If this dependency is fulfilled for a domain, which clearly is the case for the value domain or the domain of predicate abstraction, then this algorithm can be applied there. Furthermore, note that the infeasible sliced prefixes computed by Algorithm 6 have some interesting characteristics:

1. Each infeasible sliced prefix φ starts with the initial operation op_1, and ends with an assume operation contradicting the previous operations of φ, i. e., $\widehat{\mathsf{SP}}_\varphi(\top) = \bot$.

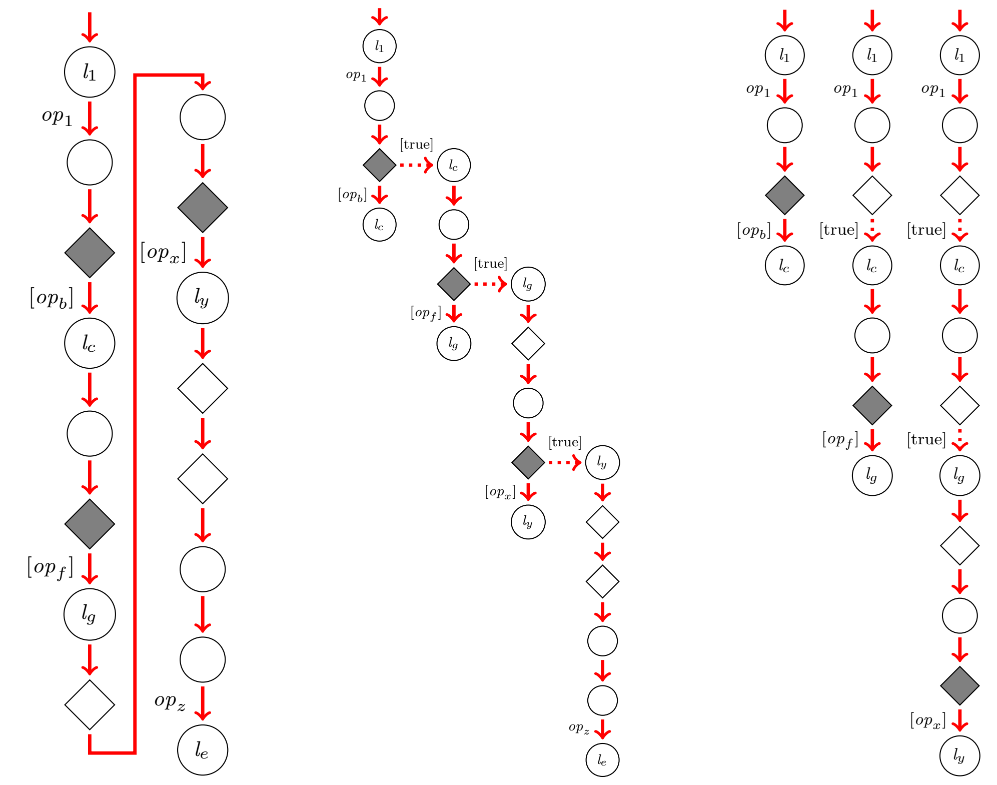
2. The i-th infeasible sliced prefix, excluding its (final and only) contradicting assume operation and location, is a prefix of the $(i + 1)$-st infeasible sliced prefix.

3. All infeasible sliced prefixes differ from a prefix of the original infeasible error path σ only in their no-op operations.

The visualizations in Figure 7.2 capture the details of applying Algorithm 6 on an infeasible error path $\sigma = \langle (op_1, l_1), \ldots, (op_z, l_e) \rangle$. Figure 7.2a shows the original error path σ. Nodes represent program locations and edges represent operations between these locations. The operation are either assignments to program variables or assume operations over program variables. To allow easier distinction, program locations that are followed by assume operations are drawn as diamonds, while other program locations are drawn as squares. Program locations before contradicting assume operations are drawn with a gray background. The sequence of operations ends in the target state, denoted by l_e. Figure 7.2b depicts the cascade of sliced prefixes that the algorithm builds during its progress. Figure 7.2c shows the three infeasible sliced prefixes that Algorithm 6 returns for this example.

The refinement procedure can now use any of these infeasible sliced prefixes to create interpolation problems, and is not bound to a single, specific interpolant sequence for the original infeasible error path. A selection of refinements from different precisions is now possible. The following lemma states that this is a valid refinement process.

Lemma 7.4.1. *Let σ be an infeasible error path and φ be the i-th infeasible sliced prefix for σ that is extracted by Algorithm 6, then all interpolant sequences for φ are also interpolant sequences for σ.*



(a) Infeasible error path (b) Cascade of sliced prefixes (c) Infeasible sliced prefixes

Figure 7.2: From one infeasible error path to a set of infeasible sliced prefixes

Proof. Let $\sigma = \langle(op_1, l_1), \ldots, (op_n, l_n)\rangle$ and $\varphi = \langle(op_1, l_1), \ldots, (op_w, l_w)\rangle$. Let Γ_{φ^j} be the j-th interpolant of an interpolant sequence for φ, i. e., for the two constraint sequences $\gamma^-_{\varphi^j} = \langle op_1, \ldots, op_j\rangle$ and $\gamma^+_{\varphi^j} = \langle op_{j+1}, \ldots, op_w\rangle$, with $1 \leq j < w$. Because φ is infeasible, the two constraint sequences $\gamma^-_{\varphi^j}$ and $\gamma^+_{\varphi^j}$ are contradicting, and therefore, Γ_{φ^j} exists (cf. Lemma 4.6.2). The interpolant Γ_{φ^j} is also an interpolant for $\gamma^-_{\sigma^j} = \langle op_1, \ldots, op_j\rangle$ and $\gamma^+_{\sigma^j} = \langle op_{j+1}, \ldots, op_n\rangle$, if the following is valid:

1. the implication $\gamma^-_{\sigma^j} \implies \Gamma_{\varphi^j}$ holds,
2. the conjunction $\Gamma_{\varphi^j} \wedge \gamma^+_{\sigma^j}$ is contradicting, and
3. the interpolant Γ_{φ^j} contains only variables that occur in both $\gamma^-_{\sigma^j}$ and $\gamma^+_{\sigma^j}$.

Consider that $\gamma^-_{\varphi^j}$ was created from $\gamma^-_{\sigma^j}$ by replacing some assume operations by no-op operations, and that $\gamma^+_{\varphi^j}$ was created from $\gamma^+_{\sigma^j}$ by replacing some assume operations by no-op operations and by removing the operations $\langle op_{w+1}, \ldots, op_n\rangle$ at the end. Thus, both $\gamma^-_{\varphi^j}$ and $\gamma^+_{\varphi^j}$ do not contain any additional constraints (except for no-op operations) than $\gamma^-_{\sigma^j}$ and $\gamma^+_{\sigma^j}$, respectively.

Because Γ_{φ^j} is an interpolant for $\gamma^-_{\varphi^j}$ and $\gamma^+_{\varphi^j}$, we know that $\gamma^-_{\varphi^j} \implies \Gamma_{\varphi^j}$ holds, and because $\gamma^-_{\sigma^j}$ can only be stronger than $\gamma^-_{\varphi^j}$, Claim (1) follows. The conjunction $\Gamma_{\varphi^j} \wedge \gamma^+_{\varphi^j}$ is contradicting, and $\gamma^+_{\sigma^j}$ can only be stronger than $\gamma^+_{\varphi^j}$. Thus, Claim (2) holds. Because Γ_{φ^j} references only variables that occur in both $\gamma^-_{\varphi^j}$ and $\gamma^+_{\varphi^j}$, which do not contain more variables than $\gamma^-_{\sigma^j}$ and $\gamma^+_{\sigma^j}$, respectively, Claim (3) holds. □

7.5 Refinements over Infeasible Sliced Prefixes

Extracting good precisions from infeasible error paths is key to the CEGAR technique, and the choice of interpolants influences the quality of the precision and the abstract model, and thus, ultimately determines the effectiveness of the verification process. Based on the results from the previous section, the refinement procedure is now able to control, by selecting a precision that is derived from an available infeasible sliced prefix, how the abstract model may evolve.

This is possible because ExtractSlicedPrefixes (cf. Algorithm 6) extracts from a given infeasible error path not only one single interpolation problem for obtaining a refined precision, but a set of (more abstract) infeasible sliced prefixes and thus, a set of interpolation problems, from which a refined precision can be extracted. The interpolation problems for the extracted paths can be given, one by one, to the interpolation engine, in order to derive interpolants for each infeasible sliced prefix individually. Hence, the refinement component of the analysis is no longer dependent on what the interpolation engine produces, but instead it is free to choose from a set of interpolant sequences the one that it finds most suitable. The move from solving a single interpolation problem to solving multiple interpolation problems, and understanding the selection of refinements as an optimization problem, is a key insight of our novel approach.

Algorithm Refine$^+$ (cf. Algorithm 7), which represents an extension of Algorithm Refine (cf. Algorithm 4), may be plugged in the CEGAR algorithm (cf. Algorithm 2), where, instead of using an infeasible program path directly for a standard interpolation-based refinement, we can now use the new module Refine$^+$, because the latter can substitute the refinement procedure in analyses based on CEGAR.

This new module first calls ExtractSlicedPrefixes to extract a set of infeasible sliced prefixes, which are more abstract than the original program path. Second, Refine$^+$ calculates the precision for each infeasible sliced prefix using a regular refinement procedure, e. g., Refine, and stores the pairs in the set τ. Third, the algorithm selects a refinement from τ. This selection is implemented in a method SelectRefinement and can access various details of the precisions, e. g., which variables are referenced in the precision. Each implementation of SelectRefinement, i. e., each heuristic, receives

Algorithm 7: Refine$^{+}(\sigma)$

```
Input      : an infeasible error path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$
Output     : a precision $\pi \in \Pi$
Variables  : a set $\Sigma$ of infeasible sliced prefixes of $\sigma$,
             a set $\tau$ of pairs of an infeasible sliced prefix and a precision
1 $\Sigma$ := ExtractSlicedPrefixes($\sigma$)
  // compute refinement for each infeasible sliced prefix, using
     Algorithm 4
2 foreach $\varphi_j \in \Sigma$ do
3   | $\tau := \tau \cup \{(\varphi_j, \mathsf{Refine}(\varphi_j))\}$
  // select a refinement based on the original path, the infeasible
     sliced prefix, and their respective precision
4 return SelectRefinement($\sigma, \tau$)
```

as input the original infeasible path as well as the set of all pairs of infeasible sliced prefix and respective precisions.

The remainder of this chapter sheds some light on the characteristics and the potential of selecting refinements, before the next chapter presents heuristics that can be used for proper implementations of the procedure SelectRefinement.

7.6 Evaluation

In the previous section we presented a technique for extracting from a single infeasible error path a set of infeasible sliced prefixes, where each of these infeasible sliced prefixes can be used for performing a refinement, thus ultimately enabling the selection of refinements.

There are two major requirements that need to be fulfilled such that selection of different infeasible sliced prefixes can actually have an effect.

1. During the course of the verification process there must be at least one refinement where more than one infeasible sliced prefix is available, otherwise no actual selection is possible.

2. The interpolation engines must be able to derive different interpolants and precisions from different infeasible sliced prefixes. Only this way it is possible to compute a precision from one infeasible sliced prefix that then would lead to another, ideally better abstraction than when computing a precision using the original infeasible error path.

In order to show that selecting different infeasible sliced prefixes actually has an effect we performed an evaluation utilizing Algorithm ExtractSlicedPrefixes to extract infeasible sliced prefixes. For the evaluation we again used the same experimental setup as before, e. g., as in Sections 3.5, also relying on BenchExec the same way as before.

7.6.1 Infeasible Sliced Prefixes for the Value Analysis

In a first evaluation we examine the effects of selecting different infeasible sliced prefixes for the value analysis. We present here the characteristics for two simplistic selection heuristics, namely Length-Min and Length-Max. These select from a set of infeasible sliced prefixes the shortest and longest infeasible sliced prefix, respectively. Several characteristics of the application of these two selection heuristics are presented in Table 7.1.

In order to allow reproducibility of the evaluation, an example for a complete command line for applying refinements over infeasible sliced prefixes to the value analysis as well as the full results and raw data are available on our supplementary web page [1].

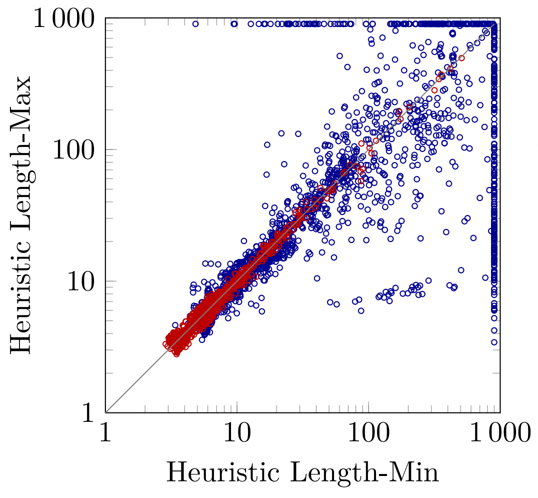
Note that for 1 274 of the 4 283 verification tasks there is no selection of infeasible sliced prefixes possible (cf. row no selection), i. e., in every refinement there was at most one infeasible sliced prefix to choose from. However, for 481 of these verifications tasks, there is no refinement needed at all (cf. row no refinements), and for another 702 verifications tasks a single refinement is enough for both selection heuristics to come to a verdict. So we note that selection of infeasible sliced prefixes is possible in most cases and we conclude that requirement 1 from above is fulfilled.

The same is true for requirement 2. For the categories DeviceDriversLinux64, ECA, ProductLines, and Sequentialized the two approaches show significant differences in verification effectiveness (cf. row solved$_{\text{Min}}$ and solved$_{\text{Max}}$). In addition, for the set of commonly solved verification tasks (cf. row solved$_{\text{Com}}$) the two approaches differ considerably in CPU time (cf. rows CPU time$_{\text{Min}}$ and CPU time$_{\text{Max}}$) and number of iterations (cf. rows iterations$_{\text{Min}}$ and iterations$_{\text{Max}}$) needed for solving the respective verification tasks. For example, relying on the selection heuristic Length-Min for category BitVectorsReach, one can save around 33 % of the CPU time compared to when using the heuristic Length-Max. Yet, for category ProductLines it is the other way round, because when using the selection heuristic Length-Max the verification process only takes 38 % of the CPU time compared to when using heuristic Length-Min. In addition, the number of iterations differs by an order of magnitude. With the selection heuristic Length-Max a total of 14 million iterations are needed, with heuristic Length-Min it takes up to 100 million iterations.

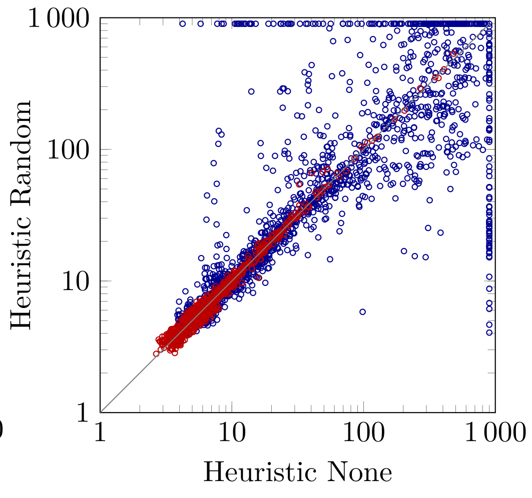
[1] `http://www.sosy-lab.org/research/phd/loewe/#InfeasibleSlicedPrefixesVa`

	BitVectorsReach	ControlFlow	DeviceDriversLinux64	ECA	Floats	Loops	ProductLines	Sequentialized	Simple	Overall
total	48	48	2 120	1 140	81	141	597	62	46	4 283
no selection	23	11	1 031	0	80	93	0	0	36	1 274
no refinements	15	2	400	0	14	43	0	0	7	481
solved$_{\text{Min}}$	46	45	1 651	497	81	112	467	38	46	2 983
solved$_{\text{Max}}$	46	45	1 773	516	80	115	361	31	46	3 013
solved$_{\text{Com}}$	46	45	1 622	458	80	112	356	29	46	2 794
CPU time$_{\text{Min}}$ (h)	0.12	0.26	14	16	0.082	0.15	2.6	0.43	0.16	34
CPU time$_{\text{Max}}$ (h)	0.18	0.26	11	16	0.083	0.15	0.99	0.22	0.15	29
iterations$_{\text{Min}}$ ($\times 10^6$)	0.32	1.8	460	850	0.0038	0.082	100	21	1.6	1 400
iterations$_{\text{Max}}$ ($\times 10^6$)	0.61	1.9	310	810	0.0038	0.082	14	11	1.5	1 100
refinements$_{\text{Min}}$	96	447	5 417	4 108	83	115	3 546	588	323	14 723
refinements$_{\text{Max}}$	104	461	4 330	3 548	83	113	1 520	535	300	10 994
avg. prefixes$_{\text{Min}}$	2.4	1.8	4.7	170	1	2.2	6.9	13	1.7	52
avg. prefixes$_{\text{Max}}$	14	2.5	5.5	320	1	3.5	5.9	19	2.1	110

Table 7.1: Table showing the effects of two different selection heuristics for the value analysis



(a) Length-Min vs. Length-Max



(b) None vs. Random

Figure 7.3: Scatter plots comparing the CPU time of the value analysis using different heuristics for selecting infeasible sliced prefixes

As both approaches only differ in the selection heuristic, this means that depending on the selection heuristic different interpolants are found from which different precision are extracted, which sometimes are better and sometimes are worse for obtaining a concise abstract model. Note also that the number of refinements needed by both approaches differ, and that the heuristic Length-Max needs less refinements for the commonly solved instances in the categories DeviceDriversLinux64 and ECA, plus being able to solve considerably more verification tasks there. This indicates that for these classes of verification tasks the heuristic Length-Max is well suited to find good abstractions. For the category ProductLines it is again the other way round, there Length-Min performs far better in terms of solved instances, but interestingly, as stated above, for the commonly solved instances Length-Min needs considerably more CPU time and iterations than with the heuristic Length-Max.

The differences between different heuristics, in regard to verification effectiveness, can also be seen from the scatter plots in Figure 7.3. In the left one, we compare the CPU time of the heuristic Length-Min with the CPU time of the heuristic Length-Max. In the right one, we compare the CPU time when not using a heuristic, i. e., we pass the original error path to the interpolation procedure, with the CPU time of the heuristic Random, which randomly picks one of the available infeasible sliced prefixes, and this infeasible sliced prefix is then passed to the interpolation procedure. In both plots, there are hundreds of data points positioned along the border of the respective plot, which indicates that for such a verification task a verdict can be obtained only by using one of the two heuristics. In the plot the 1 274 verification tasks where no selection of infeasible sliced prefixes is possible are highlighted in color red. Basically all of them are aligned along the diagonal, i. e., the plot underlines that the implementation of ExtractSlicedPrefixes (cf. Algorithm 6) does not cause any relevant overhead for the 1 274 verification tasks where no selection of infeasible sliced prefixes is possible.

After having reported on the effects of selecting infeasible sliced prefixes for the value analysis, we now turn our attention to the predicate analysis, and report on the effect that selecting infeasible sliced prefixes has in that domain.

7.6.2 Infeasible Sliced Prefixes for the Predicate Analysis

Same as for the value analysis, we want to know if a meaningful selection of infeasible sliced prefixes is possible also for the predicate analysis, i. e., we want to find out if the requirements 1 and 2 from above are fulfilled. For this evaluation we use the predicate analysis of CPAchecker—configured to perform single-block encoding (SBE) [15]—for which we implemented the procedures ExtractSlicedPrefixes and Refine$^+$, i. e., Algorithms 6 and Algorithms 7, as well. We use the same experimental setup as before (cf. Section 3.5), and for gaining first insights of selecting infeasible sliced prefixes for the predicate analysis we again use the selection heuristics Length-Min and Length-Max.

In order to allow reproducibility of the evaluation, an example for a complete command line for applying refinements over infeasible sliced prefixes to the predicate analysis in a SBE configuration as well as the full results and raw data are available on our supplementary web page [2].

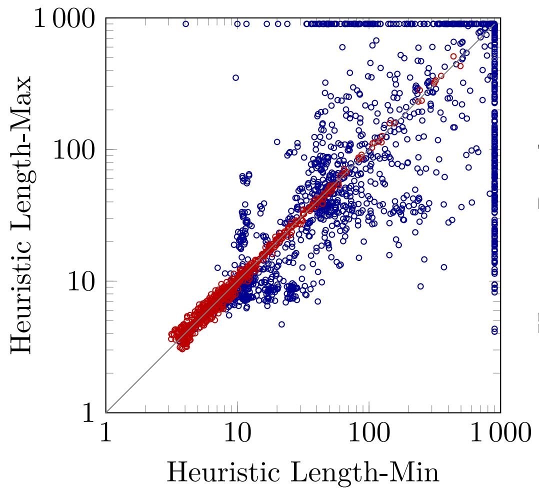
The results of this evaluation are summarized in Table 7.2, and from that, we can draw similar conclusions as for the value analysis. For 1 120 of the 4 283 verification tasks no selection is possible — but from those a total of 440 verification tasks are solved without requiring any refinement at all and another 581 verification tasks are solved with only a single refinement.

So, same as for the value analysis, in those cases where several refinements are needed for solving a verification task, there selection of infeasible sliced prefixes is possible in most cases, meaning that requirement 1 is fulfilled. According to the results in Table 7.2, requirement 2 is also fulfilled, because for the categories ControlFlow, DeviceDriversLinux64, ECA, ProductLines, and Sequentialized, there are significant differences for the two selection heuristics Length-Min and Length-Max, in the number of solved verification tasks (cf. rows solved$_{\text{Min}}$ and solved$_{\text{Max}}$) and in the number of refinements (cf. rows refinements$_{\text{Min}}$ and refinements$_{\text{Max}}$). For the set of commonly solved verification tasks (cf. row solved$_{\text{Com}}$), e. g., in the categories

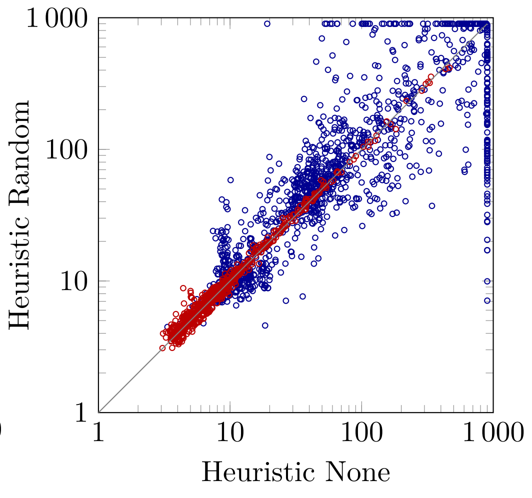
[2] `http://www.sosy-lab.org/research/phd/loewe/#InfeasibleSlicedPrefixesSbe`

	BitVectorsReach	ControlFlow	DeviceDriversLinux64	ECA	Floats	Loops	ProductLines	Sequentialized	Simple	Overall
total	48	48	2 120	1 140	81	141	597	62	46	4 283
no selection	20	9	962	0	71	58	0	0	0	1 120
no refinements	7	2	397	0	14	20	0	0	0	440
solved$_{\text{Min}}$	41	43	1 477	189	73	96	365	27	44	2 355
solved$_{\text{Max}}$	39	31	1 565	215	73	90	332	20	43	2 408
solved$_{\text{Com}}$	39	31	1 426	162	73	88	292	20	43	2 174
CPU time$_{\text{Min}}$ (h)	0.23	0.89	13	9.5	0.099	0.53	2.3	0.79	1.2	29
CPU time$_{\text{Max}}$ (h)	0.58	1.3	10	7.1	0.097	0.61	2.9	1.1	2.3	26
iterations$_{\text{Min}}$ ($\times 10^6$)	0.14	0.81	92	41	0.0028	0.11	1.4	0.69	1.1	140
iterations$_{\text{Max}}$ ($\times 10^6$)	0.55	1.3	61	28	0.0029	0.12	5.1	1.9	2.9	100
refinements$_{\text{Min}}$	239	628	4 353	1 856	73	605	2 376	428	940	11 498
refinements$_{\text{Max}}$	268	659	3 104	1 523	72	599	1 380	427	1 042	9 074
avg. prefixes$_{\text{Min}}$	2.7	2.4	2.8	46	1.2	1.7	5.7	14	3.2	11
avg. prefixes$_{\text{Max}}$	7.9	6.7	3.5	50	1.2	6.8	9	17	11	14

Table 7.2: Table showing the effects of two different selection heuristics for the predicate analysis



(a) Length-Min vs. Length-Max



(b) "None" vs. "Random"

Figure 7.4: Scatter plots comparing the CPU time of the predicate analysis using different heuristics for selecting infeasible sliced prefixes

DeviceDriversLinux64 or ECA, there are also massive differences in CPU time and number of iterations between the selection heuristics Length-Min (cf. rows CPU time_{Min} and $\text{iterations}_{\text{Min}}$) and Length-Max (cf. rows CPU time_{Max} and $\text{iterations}_{\text{Max}}$). Following the argumentation from above the two different heuristic are capable of identifying different interpolants, such that different abstract models are built, and the predicate analysis performs differently depending on the selection heuristic being employed (cf. column Overall). Again, the two scatter plots in Figure 7.4 emphasize this fact even more, because in both plots there are many data points aligned along the top or right border, which indicates that each verification task associated with such a data point can be solved using the one heuristic but cannot be solved using the other. Note that the selection of infeasible sliced prefixes does not impose any noticeable overhead for the predicate analysis either. We can show this by looking at those verification tasks for which no selection is possible — again highlighted in color red, and again aligned along the diagonal. Mind that this is not granted per se, after all, in Algorithm ExtractSlicedPrefixes, we check during each refinement for an error path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ up to n times whether or not $\widehat{\mathsf{SP}}_{\sigma_f \wedge (op,l)}(\top) = \bot$ holds. For the predicate analysis this means we have to perform up to n satisfiability checks over formulae with length 1 to n. This can only be done in a timely manner by SMT solvers that support efficient incremental solving [41], otherwise our approach based on the extraction of infeasible sliced prefixes would not scale for the predicate analysis. Fortunately, this is the case and from our evaluation we can draw the conclusion that also for the predicate analysis, using SBE, refinements for different infeasible sliced prefixes lead to significant differences in the verification effectiveness of the overall analysis.

7.6.3 Infeasible Sliced Prefixes with Large-Block Encoding

The above evaluation for the predicate analysis does not discuss the effects of selecting infeasible sliced prefixes when combined with large-block encoding (LBE) [15]. Other than with SBE [15] where such a block $b = \langle (op, l) \rangle$ always only spans over a single pair of an operation and a location, with LBE such a block $b = \langle (op_1, l_1), \dots, (op_n, l_n) \rangle$ may span over multiple pairs of an operation and a location, i. e., a block may span over the statements of whole functions or loops, and may contain an arbitrary number of assume operations, and therefore, may represent an arbitrary number of *different* paths through the respective block.

At first it is unclear how extracting infeasible sliced prefixes could be combined with LBE, i. e., how to efficiently run Algorithm 6 on a path that is large-block encoded, because such a “path” in fact may encode many different paths, and simply picking a single representative from this set of paths is not a valid strategy, because there is no guarantee that an infeasible sliced prefix extracted from such a representative is also an infeasible sliced prefix for all the other paths encoded by the block. Using all encoded paths as input for Algorithm 6 might not scale, and it remains unclear how one would combine the resulting infeasible sliced prefixes into an *infeasible sliced block prefix*.

In consideration of the rather complex and unclear approaches sketched above, we refrain from exploring any of these in this thesis in favor of a simpler approach that we briefly outline in the following. First off, note that efficient satisfiability checks are also possible with ABE [24], i. e., today’s SMT solvers are able to efficiently answer queries whether a single (large) block or a sequence of (large) blocks is contradicting or not. Furthermore, if we take a step back and realize that a path encoded with SBE also consists of blocks —blocks that only span over a single statement— then, from a theoretical point of view, it is rather straight-forward to apply the notion of infeasible sliced prefixes to ABE, too. We present a variant of the original algorithm (cf. Algorithm 6) in Algorithm ExtractSlicedBlockPrefixes (cf. Algorithm 8), which also supports ABE. Instead of iterating over the pairs (op, l) of an operation and a location of a path σ and deciding whether the current sliced prefix extended by the current element is feasible or not, it iterates over the blocks of a path σ, in order from the first to the last block, and decides whether the current sliced block prefix is feasible or not when extended by the next block of the path. Same as in the original algorithm, if the resulting sliced block prefix is infeasible, a

Algorithm 8: ExtractSlicedBlockPrefixes(σ)

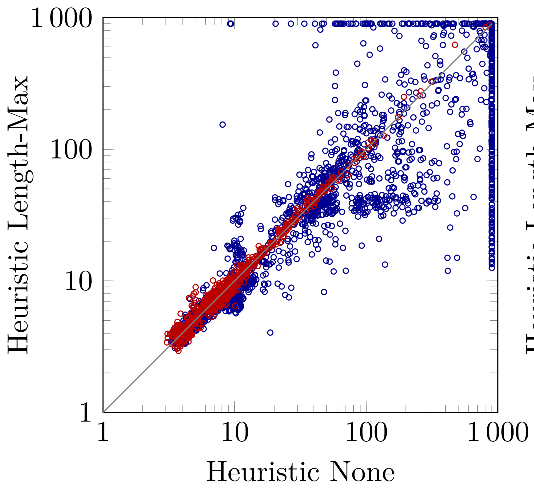
```
Input     : an infeasible block-encoded path $\sigma = \langle b_1, \ldots, b_m \rangle$, in SSA form
Output    : a non-empty set $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ of infeasible sliced block prefixes
Variables : a path $\sigma_f$ that is always feasible
1 $\Sigma := \emptyset$
2 $\sigma_f := \langle\rangle$
3 foreach $b = \langle (op_k, l_k), \ldots, (op_p, l_p) \rangle \in \sigma$ do
4     if $\widehat{\mathsf{SP}}_{\sigma_f \wedge b}(\top) = \bot$ then
          // add $\sigma_f \wedge b$ to set $\Sigma$ of infeasible sliced block prefixes
5         $\Sigma := \Sigma \cup \{\sigma_f \wedge b\}$
          // append block of no-ops
6         $\sigma_f := \sigma_f \wedge \langle ([true], l_k) \wedge \ldots \wedge ([true], l_p) \rangle$
7     else
          // append original block
8         $\sigma_f := \sigma_f \wedge b$
9 return $\Sigma$
```

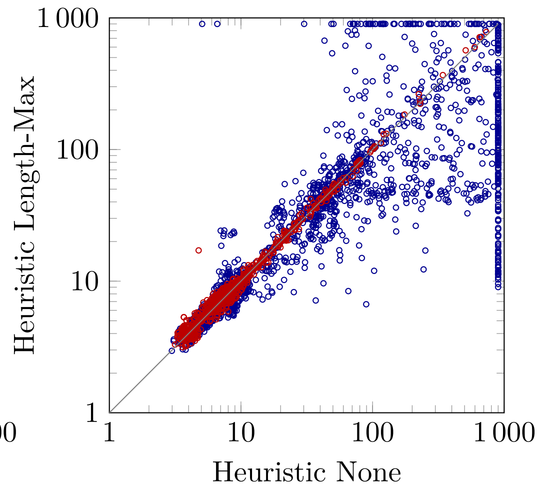
new infeasible sliced block prefix is added to the set of infeasible sliced block prefixes, and the current block is replaced by a block representing no-op operations.

It is important to note that Algorithm 8 is correct, despite the fact that not only contradicting assume operations are replaced by no-op operations but *all* operations contained in the replaced block are made ineffective by no-op operations. This might appear wrong at first, as this potentially removes assignment operations from the path. But hold in mind that the path is in static single assignment form, so an assume operation over a program variable written in an earlier, removed block can never lead to a contradiction or an infeasible sliced block prefix, because the respective program variable referenced in the assume operation appears as it would be unassigned.

The insight of thinking in blocks instead of pairs of operations and locations allows us to develop a single variant of Algorithm 8 which works for any block-encoding strategy, because different block-encoding strategies only affect the size of the resulting blocks, which is not relevant for our variant of Algorithm 8.

Note however, that the larger the block size is, the less infeasible sliced block prefixes can be extracted for a given infeasible path σ. This is because, first, the larger the blocks, the fewer blocks are available for a given path, and consequently the chance for extracting infeasible sliced block prefixes decreases. Imagine the extreme case where a path is encoded with just a single block, in which case no extra infeasible sliced block prefix different from the input can be extracted. Second, in case an infeasible sliced block prefix is found, the current block is replaced by a



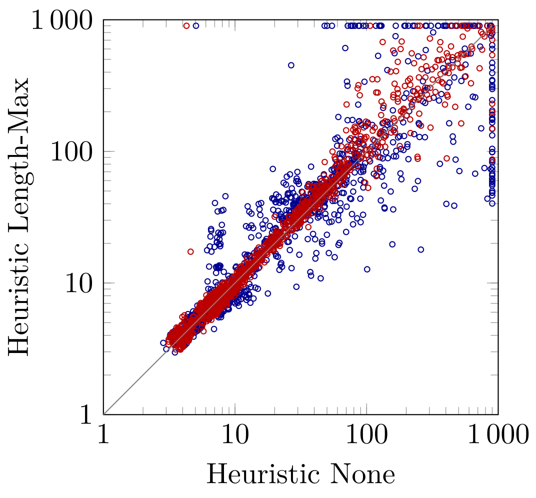


(a) None vs. Length-Max with SBE

(b) None vs. Length-Max with ABE-lj

(c) None vs. Length-Max with ABE-lf

(d) None vs. Length-Max with ABE-l



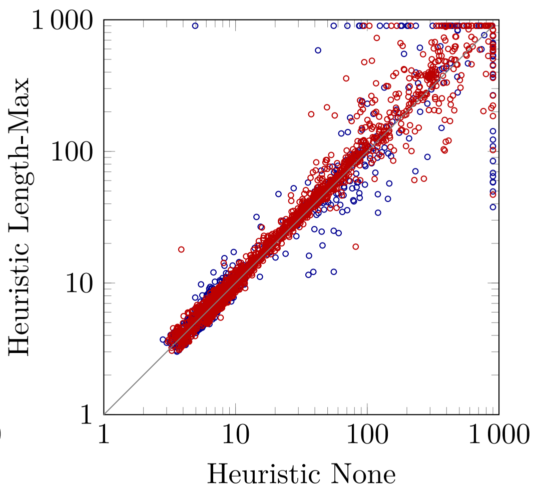


Figure 7.5: Scatter plots comparing the CPU time of the predicate analysis if using the selection heuristic Length-Max and not explicitly using a selection heuristic for different ABE-block sizes

block of no-op operations, and with that, no statement in the replaced block may lead to a contradiction with a following block. Again, the larger the blocks are, the more statements are replaced by no-op operations, and that lowers the chance for extracting infeasible sliced block prefixes along the remaining path.

Concretely, if we apply ABE and let blocks span over whole functions or loops, we expect less of an effect of refining over infeasible sliced block prefixes than compared to if using it with SBE, simply because with large blocks there are less infeasible sliced block prefixes available per refinement.

In order to allow reproducibility of the evaluation, an example for a complete command line for applying refinements over infeasible sliced prefixes to the predicate analysis in an ABE configuration as well as the full results and raw data are available on our supplementary web page[3].

In Figure 7.5 we show a comparison of the selection heuristic Length-Max with the case where no selection heuristic is explicitly set for four different block-sizes, namely for SBE (cf. Figure 7.5a), where a new block starts at each operation, for ABE-lj (cf. Figure 7.5b), where a new block starts at loop heads or whenever control flow joins, for ABE-lf (cf. Figure 7.5c), where a new block starts at loop heads or at function entries or exits, and for ABE-l (cf. Figure 7.5d), where a new block starts at loop heads, only. As before, data points for verification tasks where only a single infeasible block prefix is available, and thus no selection being possible, are drawn in red color. With that one can see easily that larger blocks often prohibit a selection process, as way more data points are drawn in red color in the plots for ABE-lf and ABE-l. Another interesting observation from these two plots is, that many of the verification task where no selection process is possible are no longer aligned along the diagonal, i. e., despite no selection being possible, the extraction of infeasible sliced block prefixes influences the efficiency of the verification process. This is not because the extraction of infeasible sliced block prefixes would introduce an overhead —after all, the red data points spread about equally on both sides of the diagonal— but with the extraction of an infeasible sliced block prefix and the replacement of the respective contradicting block, it may happen that another contradiction present in the original path is no longer present in the path after having extracted the infeasible sliced prefix.

We demonstrate the last point by an example verification task taken from the official SV-COMP'16 repository (cf. Figure 7.6). In Figure 7.6a we present the source code of the verification task, and in Figure 7.6b a possible error path of the verification task is shown, where each horizontal line denotes the start of a new ABE-block. For this example we refer to an ABE-lf configuration, i. e., a new block starts after each function entry and exit (lines 2 and 15 in Figure 7.6b) and whenever a loop head is passed (lines 4, 8 and 12 in Figure 7.6b). The shown path contains two contradictions, first `[!( y < 1024)]` does not hold, and second, `[cond == 0]` does not hold, because `int x` is always 0 at this location, so `int cond` always equals 1 here. For the first contradiction we can only find interpolants referencing the loop-counter variable `int y`, and with that, the loop gets unrolled iteration after

[3]`http://www.sosy-lab.org/research/phd/loewe/#InfeasibleSlicedPrefixesAbe`

```
extern void VERIFIER_error();
void VERIFIER_assert(int cond) {
  if (!(cond)) {
    ERROR: VERIFIER_error();
  }
  return;
}

int main(void) {
  unsigned int x = 1;
  unsigned int y = 0;
  while (y < 1024) {
    x = 0;
    y++;
  }

  VERIFIER_assert(x == 0);
}
```

(a) Source code of verification task

```
 1 main();

 2   x = 1;
 3   y = 0;

 4     [y < 1024]
 5     x = 0;
 6     y = y + 1;

 8     [y < 1024]
 9     x = 0;
10     y = y + 1;

12     [!(y < 1024)]
13   VERIFIER_assert((x == 0)
14     ? cond = 1
15     : cond = 0);

15     [cond == 0]
16     VERIFIER_error();
```

(b) Error path over two loop iteration

Figure 7.6: Verification task `const_true-unreach-call1.c` taken from the official SVCOMP'16 repository, and a possible infeasible error path when analyzing the task with ABE-lf

iteration. The interpolants for the second contradiction would be of great value, because they assure that `int x` is 0 and `int cond` is 1 after at least one loop iteration, which is enough to prove the verdict of the verification task being `true`.

However, if we extract infeasible sliced block prefixes as proposed by Algorithm 8 then we are not able obtain interpolants for that second contradiction mentioned above. This is because, according to Algorithm ExtractSlicedBlockPrefixes, a block that leads to a contradiction has to be replaced by no-op operations, such that the path σ_f remains feasible for future feasibility checks. In the example, the first block b that leads to the infeasibility of $\widehat{\mathsf{SP}}_{\sigma_f \wedge b}(\top)$ is the block from lines 12 to 15. Yet, with replacing the operations of this block with no-op operations there is no longer a contradiction in the path σ_f for the program variable `int cond`, because the (conditional) assignment `cond = 1` has been replaced by a no-op operation. Consequently, we can only extract one single infeasible sliced block prefix for this infeasible error path, hence, no selection of infeasible sliced block prefixes is possible, and for the one infeasible sliced block prefix we obtain, the resulting interpolants lead to repeated unrollings of the loop, forcing the analysis to exhaust all available resources.

This example explains why with larger block sizes the selection heuristics may have an influence on the verification efficiency, despite the fact that no selection is even possible. The example we refer here to can also be easily spotted in Figure 7.5c —it is the red data point on the far left aligned at the top border— i. e., it is solved in under 10 s without an explicit selection heuristic, and despite no selection is possible —as there always is just one infeasible sliced block prefix— if the selection heuristic is applied, the verification task cannot be solved within the timeout. We make similar effects responsible for those cases where significant performance differences exist between using no explicit selection heuristic and using a selection heuristic in combination with larger blocks (cf. Figure 7.5). Minor modifications to Algorithm ExtractSlicedBlockPrefixes, e. g., iterating in reverse over the blocks of the infeasible error path, solve this problem for our example verification task, however, this would not be a general solution, and we did not investigate this any further.

In conclusion, if the predicate analysis is configured to use larger blocks, then the effect of refinements over different infeasible sliced block prefixes is not as drastic when compared to running the predicate analysis with SBE.

7.6.4 Further Applications of Infeasible Sliced Prefixes

The main benefit that we think we can gain from infeasible sliced prefixes is the ability to perform *guided* refinement selection, i. e., the ability to choose favorable refinements while avoiding refinements that would make the analysis diverge, a technique that is covered in more detail during the next chapter.

Besides that, a few other techniques that are discussed in this work may benefit from infeasible sliced prefixes, such as for example, static refinement (cf. Section 5.6.1). The main problem with static refinement is, that the program variables referenced in the use-def chain of all assume operations are added to the precision increment. With the introduction of infeasible sliced prefixes, there is always at most a single infeasible assume operation left, and so it suffices to add the program variables contained in the use-def chain of exactly this infeasible assume operation, which often allows for a more compact precision increment.

The combination of infeasible sliced prefixes and the use-def-chain allows also to speed up the value interpolation (cf. Section 4.6). This is because a value interpolation query has only be performed for those operations that are referenced in the use-def chain of the single infeasible assume operation which is left in the infeasible sliced prefixes, because all other operations in the infeasible sliced prefix cannot influence the feasibility of the infeasible sliced prefix.

Furthermore, the same way infeasible sliced prefixes are applicable to the value or the predicate analysis, there is also work [25] that reports great success for an analysis based on symbolic execution (cf. Section 5.5). Beyond that, infeasible sliced prefixes are also applicable to Impact (cf. Section 5.6.3), and may be used to control the refinement process of global refinement (cf. Section 5.6.2). There, usually not just a single target state exists but multiple, forming a tree-like structure (cf. Figure 5.4). With infeasible sliced prefixes one may now try to compute refinements in such a way that each infeasible error path is refuted with a different precision increment, or all with the same, or try to find a refinement where the pivot state is shallow to refute all infeasible error paths with a single refinement, or the other way round, to have pivot states that are as deep as possible such that each infeasible error path is refuted with its own refinement.

The respective selection heuristics that actually allow such a fine-grained control over the refinement process are presented after a brief summary of this chapter.

7.7 Conclusion

In this chapter we introduced the notion of infeasible sliced prefixes, and we presented an algorithm that extracts from a single infeasible error path a set of infeasible sliced prefixes. Along with that, we provided a proof that any of these infeasible sliced prefixes can be used to perform a refinement of the analysis such that the original infeasible error path is excluded from subsequent state-space explorations. This result allows us to formulate a novel refinement procedure Refine^+ (cf. Algorithm 7) where we can now select a particular refinement from a set of available refinements, each computed from a different infeasible sliced prefix. This makes the selection of refinements possible, and in a first evaluation for both the value analysis and the predicate analysis we obtained a first impression of its potential.

7.7.1 Lessons Learned

Our first hypothesis was that a selection of infeasible sliced prefixes would be available for a large portion of verification tasks, and our second hypothesis was that different selections would influence the verification effectiveness for these verification tasks to some extent. The evaluation we performed confirmed both hypotheses for the value analysis and the predicate analysis, and that leads us to believe that other heuristics for selecting infeasible sliced prefixes allow us to reliably obtain a verdict for many verification tasks that so far cannot be solved by either of the two analyses.

7.7.2 Challenge

In the evaluation we performed, we only investigated rather simplistic and naive heuristics, which are not targeted towards a particular goal that would let us believe that the resulting abstract models become superior in general. The same is true for the case where no selection of infeasible sliced prefixes is performed, because there the selection is also random, in the sense that the selection adheres to the heuristics of the interpolation engine, without any chance for controlling that from the outside. Our evaluation underlines that relying on the heuristics of the interpolation engine is not the best approach, however, so far it is also totally unclear how to steer the selection of infeasible sliced prefixes such that the number of solved verification tasks can be maximized.

7.7.3 Proposition

The heuristics for selecting infeasible sliced prefixes investigated so far either pick refinements randomly, or select the refinement corresponding to the shortest or longest infeasible sliced prefix, so the whole potential of selecting refinements literally lies somewhere in between these heuristics. To improve on that, we propose novel heuristics for selecting refinements that lead to concise abstract models, such that the verification process does not diverge.

7.7.4 Solution

In this chapter we learned that different heuristics for selecting infeasible sliced prefixes may lead to different precisions and that this may have a significant impact on how the analysis eventually performs. Because the precision, computed from interpolants, strongly influences how the abstract model for a verification task evolves, the next chapter will focus on defining and evaluating heuristics for selecting refinements that try to assess the quality of refinements based on the interpolants associated with the respective refinements, so that the refinement process is *guided* towards abstract models that allow the verification process to converge in a timely manner.

8 Guided Refinement Selection

In the previous chapter we introduced refinements of infeasible sliced prefixes. This technique is an extension of the standard CEGAR approach, because instead of computing a single refinement for a single infeasible error path our novel technique computes a set of refinements for each infeasible sliced prefix. In an extra step added to the CEGAR loop, we then have the chance to select a specific refinement. The heuristics for selecting refinements we evaluated so far are rather ad hoc and simplistic, but showed that selecting different refinements has a significant impact on verification effectiveness. We now want to define and evaluate heuristics for *guided refinement selection* that are specifically geared towards optimizing verification effectiveness. In addition to that, we propose refinement selection to be used for a composite analysis, where refinement selection is not only performed within each component analysis, but also dictates, for a given infeasible error path, which of the component analyses ought to be preferred for a refinement.

8.1 Motivation

In order to avoid state-space explosion and divergence during the verification process, we need to keep the precision of the analysis as coarse as possible. Existing approaches that use interpolation to extract precision information from infeasible error paths assign a lot of choice to the interpolation engine, because infeasible error paths are often infeasible for a number of reasons, and it is left to the interpolation engine which one it chooses to form a proof of unsatisfiability. This choice influences the resulting precision, and one precision might be more suited for the further progress of the analysis than another. We noticed such differences already in the evaluation performed in the previous chapter (cf. Section 7.6), and our motivation here is to learn how to select the refinement that is best suited for the further progress of the analysis. For example, one particular verification task, namely `parport_true-unreach-call.i.cil.c` in the category Simple of SV-COMP could not be solved by the standard ABE-l configuration of the predicate analysis, because a loop in that verification task was getting unrolled continuously. However, the analysis avoided the loop unrolling and was able to solve this verification task efficiently if ABE-lf was applied as block-encoding strategy. The difference in verification efficiency for this single verification task sparked the design of novel traversal and block-encoding strategies, the latter being as sophisticated as taking clustering techniques for the CFA into account. None of these approaches improved the situation for the verification task `parport_true-unreach-call.i.cil.c` nor did they have a major influence on the verification efficiency for other SV-COMP categories. In contrast, with guided refinement selection the analysis is in fact able to select suitable refinements, and for this example verification task from above, our novel approach allows to avoid the loop unrollings both for the ABE-l and the ABE-lf case. We would like to generalize this technique to more analyses and configurations, and therefore, we perform a detailed inspection of the interpolant sequences associated with each refinement, where we, for example, try to assess the quality of a refinement by looking at the program variables being referenced in the interpolant sequences, or by estimating the effort needed for re-exploration based on the location of the pivot states associated with the interpolant sequences.

8.2 Related Work

In many programming languages the type of a variable is quite coarse and only imposes an upper limit on the range of values a variable of this respective type can be assigned to. For example, the C type **`int`** is typically used also for program variables having boolean character, i. e., program variables that are only ever assigned to either 0 or 1. For this purpose, domain types [6] have been proposed, which refine the type system of a programming language and allow to classify program variables according to their actual range or usage in a verification task. It was shown that distinguishing variables on a more fine-grained level can be beneficial for verification [6, 50, 97, 105].

Lazy abstraction [69] suggests to always prune and re-explore the state space beginning at the pivot state, i. e., at that state in the ARG that is closest to the initial state and for which the current refinement contains new precision elements, i. e., new interpolants. The advantage of this is that, compared to simply restarting the exploration with the new precision, less parts of the state space have to be pruned and re-explored. For an arbitrary verification task, it is not immediately clear if it is better to restart the exploration with the new precision, or to continue from the pivot state. With the availability of refinement selection there is now even more choice from where to prune and re-explore, because there are potentially multiple refinements to select, and each of those might have a different pivot state.

8.3 Heuristics for Guided Refinement Selection

We first focus on tuning refinement selection for a single analysis, and afterwards we propose several heuristics of which we think might lead to suitable refinements.

8.3.1 Selection by Domain-Type Score of Path Precision

Our first heuristic inspects the types of variables in the resulting precisions and prefers refinements with simpler or smaller types. With domain types [6], one can distinguish between variables that are used as booleans, variables that are used in equality relations only, in arithmetic expressions, or in bit-level operations, and variables that share characteristics of a loop counter.

Loop counters are a class of variables that a program analysis should ideally omit from the abstract model of a verification task in many cases. But because loop-counter variables occur in assume operations at the loop exit, they often relate to a reason of infeasibility of a given infeasible error path. Thus, those variables are often included in the interpolant sequence that a standard interpolation engine might produce, forcing the program analysis to track them. Therefore, a promising heuristic is to avoid precisions that contain loop counters, and prefer precisions with only program variables of a "simpler", e. g., boolean type. The rationale behind this heuristic is that variables with only a small number of different valuations have less values to grow the state space, and therefore are to be preferred. If, however, reasoning about the specification demands unrolling a loop, then the termination of the verification process may be delayed by first refining towards other, irrelevant properties of the verification task.

In order to compute the domain-type score for a precision π, we first define a function $\delta : X \mapsto \mathbb{N} \setminus \{0\}$ that assigns to each program variable its domain-type score. The domain type for all program variables can be inferred by an efficient data-flow analysis [6], and we use low score values for variables with small ranges, e. g., boolean variables, and a specifically high value for loop counters. Thus, we define the domain-type score of a precision as the product over the domain-type scores of every variable referenced in the precision: $\mathsf{DomainTypeScoreOfPrecision}(\pi, \delta) = \prod_{x \text{ referenced in } \pi} \delta(x)$.

This function, as well as the design of function δ, are mere proposals for assessing the quality of a precision. However, we experimented with several different implementations for both functions, and come to the conclusion that the most important requirement to be fulfilled is that precisions with only boolean variables should be associated with a low score, and precisions referencing loop-counter variables should be penalized with a high score.

8.3.2 Selection by Depth of Pivot Location of Path Precision

The structure of a refinement, i. e., which parts of the path and the state space are affected, can also be used for refinement selection. For example, refining close to the target state may have a different effect than refining close to the program entry. We define the *pivot location* of an infeasible error path σ as the first location in σ

where the precision is not empty. If using lazy abstraction [69], this pivot location is associated with the pivot state, i. e., the state from which on the reached state space is pruned and re-explored after the refinement. The depth of this pivot location can be used for comparing possible refinements and selecting one of them. Formally, for a precision π extracted for a path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$, the depth of the pivot location is defined as $\mathsf{PivotDepthOfPrecision}(\pi, \sigma) = \min \{i \mid \pi(l_i) \neq \emptyset\}$. Mind that the minimum always exists, because there is always at least one location with a non-empty precision.

Selecting a refinement with a deep pivot location (close to the end of the path) is similar to counterexample minimization [5]. It has the advantage that (if using lazy abstraction) only a fraction of the state space has to be pruned and re-explored, which can be more efficient for some verification tasks. Furthermore, the precision will specify to track preferably information local to the target state and thus avoid unfolding the state space in other parts of the verification task. However, preferring a deep pivot location may have negative effects if some information close to the program entry is necessary for reasoning over a verification task (e. g., initialization of global variables). Refining at the beginning of an error path might also help to rule out a large number of similar error paths with the same precision, which might otherwise be discovered and refined individually.

8.3.3 Selection by Width of Path Precision

Another heuristic that is based on the structure of a refinement is to use the number of locations in the infeasible error path for which the precision is not empty, which we define as the *width* of a precision. This corresponds to how long on a path the analysis has to track additional information during the state-space exploration, and thus correlates to how long the precision contributes to the state-space unfolding. Similarly to the depth of the pivot location, this heuristic also deals with some form of locality, but instead of using the locality in relation to the depth, it uses the locality in relation to the width. Formally, for a precision π extracted for a path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$, the width of the precision is defined as $\mathsf{WidthOfPrecision}(\pi, \sigma) = 1 + \max I - \min I$, where $I = \{i \mid \pi(l_i) \neq \emptyset\}$ is the set of indices along the path with a non-empty precision.

It may seem that narrow precisions are in general preferable, because it means tracking additional information only in a smaller part of the state space. However, narrow precisions favor loop counters because in many loops the statements for assigning to the loop counter are close to the loop-exit edges. Thus, selecting a narrow precision often leads to loop unrollings.

8.3.4 Selection by Length of Infeasible Sliced Prefix

Selecting the shortest or longest infeasible sliced prefix, respectively, are two other heuristics that are applicable for refinement selection as well. We presented these two heuristics already in the previous chapter. While both these heuristics are legitimate and may work on some benchmarks, we do not regard them as systematical, in the sense that they are not guided towards what we consider as a beneficial characteristic of a precision. However, we resort to using this heuristic as a tie-breaker, such that we choose either the refinement that is associated with the shorter or with the longer infeasible sliced prefix, in case the main selection heuristic, e. g., the one based on domain types, computes the same score for two or more refinements. Mind that this heuristic always returns one distinct refinement, and therefore is well suited as a tie-breaker.

8.3.5 Composition of Heuristics

Using the length of the infeasible sliced prefixes as tie-breaker can already be seen as a composition of selection heuristics. Furthermore, it is possible to build composite heuristics from the heuristics introduced above. For example, as the heuristic based on the width of precisions is prone to favour a precision based on loop counters as explained above, one can first select those refinements that have a precision with a low width, and from this subset select those having a low domain-type score in order to try avoiding refinements that reference loop counters in their precision. Following this schema, many compositions of selection heuristics are possible, simply by first selecting by one heuristic, then selecting from the result with the next heuristic, until only a single refinement is left, which then matches the required characteristics the closest.

8.3.6 Tailor-Made Heuristics using Domain Knowledge

Besides selecting refinements based on the characteristics of the precisions associated with the refinements, one can also design a custom selection heuristics that take domain knowledge into account. For example, we designed a heuristic for refinement selection specifically tailored to the verification tasks of the RERS challenge 2014. This significantly improved the verification effectiveness of CPAchecker and allowed it to obtain good results in that competition [1]. This shows that using domain knowledge in the refinement step of CEGAR is a promising direction, and a specific heuristic for refinement selection is a suitable place to define this.

[1]Results available at `http://www.rers-challenge.org/2014Isola/`

8.4 Evaluation of Intra-Analysis Refinement Selection

8.4.1 Configuration

We use Algorithm ExtractSlicedPrefixes (cf. Algorithm 6) to generate infeasible sliced prefixes during a refinement performed by Algorithm Refine$^+$ (cf. Algorithm 7), i. e., same as introduced in the previous chapter (cf. Section 7.4). In order to properly evaluate the effect of the precisions that are chosen by the refinement-selection heuristic, we configure the analysis to interpret the precision globally, i. e., instead of a mapping from program locations to sets of precision elements, the discovered precision elements get used at all program locations. Note that this does not change the precision as seen by the refinement-selection heuristic, but only the precision that is given to the state-space exploration. For the same reason, we also restart the state-space exploration with the refined precision from the initial program location after each refinement. Otherwise, i. e., if we used lazy abstraction and re-explored only the necessary part of the state space, not only the new precision but also the amount of re-explored state space would differ depending on the selected refinement, which would have an undesired influence on the experiment.

As before, the predicate analysis uses SmtInterpol [40] as SMT solver and interpolation engine[2], and it is configured to use single-block encoding [24], because, as shown in the evaluation during the previous chapter, for larger blocks there is less chance for actual refinement selection. Furthermore, for the value analysis a concept similar ABE is not applicable, and in order to compare the effects of guided refinement selection for the value analysis with those for the predicate analysis it makes sense to limit the predicate analysis to SBE.

Throughout the evaluation in this chapter we again use the same experimental setup as before (cf. Section 3.5), except that we executed each verification run on two instead of four CPU cores in order to run more verification tasks in parallel which allows a more timely execution of the almost 150 000 verification runs in this evaluation.

Refinement-Selection Heuristics

We experiment with implementations of the procedure SelectRefinement in Algorithm 7 based on the heuristics from Section 8.3, specifically such that it returns the precision for (1) the shortest (Length-Min) or (2) longest (Length-Max) infeasible sliced prefix, the precision with the (3) narrowest (Width-Min) or (4) widest (Width-Max) precision, with the (5) the shallowest (Depth-Min) or (6) with the

[2]With MathSAT 5 we observed similar effects if guided refinement selection is used.

deepest (Depth-Max) pivot location, or with the (7) lowest (Score-Min) or (8) highest (Score-Max) domain-type score[3].

We also experiment with compositions of heuristics, where at first a primary heuristic is asked, and if this does not lead to a unique selection, a secondary heuristic is used as a tie breaker to select one of those refinements that are ranked best by the primary heuristic. We use the composition of lowest domain-type score (Score-Min) plus narrowest precision (Width-Min) as new composite heuristic named Score & Width, and narrowest precision (Width-Min) plus lowest domain-type score (Score-Min) as the new composite heuristic Width & Score. For comparison, we report the results of a heuristic that selects refinements randomly (Random), as well as for the case where no explicit refinement selection is performed (None), i. e., where the precision extraction is based on the complete, original infeasible error path and the choice of refinements is solely left to the interpolation engine.

In all heuristics for refinement selection, if necessary, we use the length of the infeasible sliced prefix as a final tie breaker, and select from equally ranked refinements the one with the longest infeasible sliced prefix[4].

8.4.2 Refinement Selection for the Predicate Analysis

We first evaluate the presented heuristics for refinement selection when applied to the predicate analysis.

In order to allow reproducibility of the evaluation, an example for a complete command line for performing guided refinement selection for the predicate analysis as well as the full results and raw data are available on our supplementary web page[5].

In Table 8.1 we list the number of verification tasks that the predicate analysis can solve for the refinement-selection heuristics mentioned above. Numbers printed in bold digits highlight the best configuration(s) for a category, and we print the categories BitVectorsReach, Floats, Loops, and Simple in light gray, because for the configuration used in this evaluation refinement selection hardly makes a difference in these categories.

In addition, we include the two meta heuristics *Optimal* and *Oracle*. The meta heuristic *Optimal* represents, for each category, the maximum of verification tasks that one of the actual heuristics can solve, i. e., assuming we would pick the refinement-selection heuristic performing best for a given *category*, then this amount of verification tasks could be solved. The meta heuristic *Oracle* denotes, for each category, the total of verification tasks that *any* of the actual heuristics can solve, i. e., assuming

[3]We do not expect the precision with a high domain-type score to be actually useful, we report its results merely for comparison.

[4]Experiments show no relevant difference between selecting by the shortest or the longest infeasible sliced prefix in case of a tie in the primary selection heuristic.

[5]`http://www.sosy-lab.org/research/phd/loewe/#GuidedRefinementSelectionPa`

		BitVectorsReach	ControlFlow	DeviceDriversLinux64	ECA	Floats	Loops	ProductLines	Sequentialized	Simple	Overall
total		48	48	2 120	1 140	81	141	597	62	46	4 283
Length	Min	41	**43**	1 471	177	73	96	356	27	44	2 328
	Max	39	31	1 564	194	73	90	332	20	42	2 385
Width	Min	41	37	1 464	**280**	73	98	346	24	43	2 406
	Max	39	36	1 545	174	73	90	312	27	41	2 337
Depth	Min	39	34	1 526	152	73	90	**358**	26	44	2 342
	Max	41	36	1 468	206	73	97	329	24	43	2 317
Score	Min	41	40	**1592**	217	73	95	339	**29**	44	**2470**
	Max	38	23	1 459	191	73	89	302	20	43	2 238
Score & Width		41	39	1 563	214	73	98	345	**29**	44	2 446
Width & Score		41	37	1 463	279	73	98	338	24	43	2 396
Random		43	42	1 526	192	73	99	341	26	44	2 386
None		43	42	1 525	218	73	97	324	28	43	2 393
Optimal		43	43	1 592	280	73	99	358	29	44	2 561
Oracle		43	43	1 650	337	73	99	412	31	44	2 732

Table 8.1: Number of solved verification tasks for the predicate analysis with refinement selection using different heuristics

an oracle would exist that answers which of the actual heuristics presented above is best for a given *verification task*, then the respective number of verification tasks could be solved.

In addition to the tabular overview, we show in Figure 8.1 a quantile plot for the refinement-selection heuristics Score-Max, Score-Min, as well as *Oracle*, and compare it against the case where no explicit refinement selection is performed (None).

Guided Refinement Selection Matters

The most interesting fact we learn from this first evaluation is that the configuration which does not apply any explicit refinement-selection heuristic is never the best configuration for any category, including category Overall. And from Figure 8.1 one can see that the heuristics for refinement selection we presented indeed allow to guide towards different refinements that lead to significant differences in verification effectiveness. If refinements are favoured that lead to tracking information over

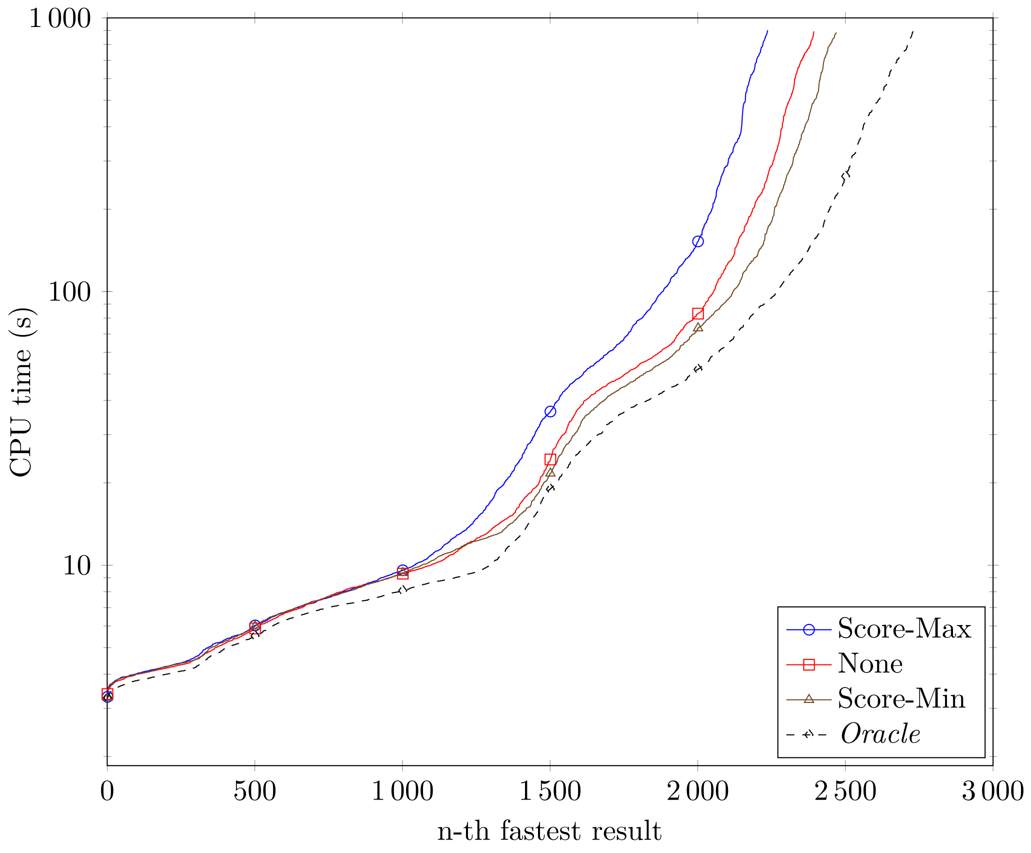


Figure 8.1: Quantile plot comparing the CPU time of predicate analysis without and with refinement selection using different heuristics

program variables with a high domain-type score (Score-Max), then the analysis performs far worse in comparison to not using any explicit refinement-selection heuristic (None). In contrast, if refinements are chosen based on the refinement-selection heuristic Score-Min, then the analysis performs clearly better overall, and, as demonstrated by the graph of refinement-selection heuristic *Oracle*, given the best refinement-selection heuristic for a verification task is known beforehand, guided refinement selection has an impressive effect on the verification effectiveness of the analysis. This shows that the heuristics of the interpolation engine —with which we are stuck without diligent refinement selection— are not always well suited for software verification, and that a deviation away from the heuristics of interpolation engine often pays off.

According to our experiment this also holds true for a predicate analysis running in an ABE-l or ABE-lf configuration, but the improvement of applying refinement selection, e. g., using heuristic Score-Min, is limited to category `DeviceDriversLinux64`, while for all other categories no real improvements are noticeable.

Discussion

As Table 8.1 shows, none of the refinement-selection heuristics works best for all classes of verification tasks, but instead, in each category a different heuristic is the best. In the following, we would like to highlight and explain a few interesting results for some categories and refinement-selection heuristics. Note that the following discussion is based on the investigation of some samples of verification tasks, and on our understanding of the characteristics of the verification tasks in the SV-COMP categories, and we do not claim that our explanations are necessarily applicable to all verification tasks.

The verification tasks of the category `DeviceDriversLinux64` contain many functions and loops, and aspects about the specification are encoded in global boolean program variables that are checked right before the target state. Hence, the heuristic Score-Min is effective because it successfully selects precisions with the “easy” and relevant boolean program variables. The heuristics Length-Max, Depth-Min, and Width-Max all happen to work well, too, because those relevant program variables are initialized at the beginning and read directly before the target state, meaning that the corresponding infeasible sliced prefix will be long, and resulting precisions containing them will be “shallow” and “wide“, as they start tracking information close to the program entry, and all the way to the target state. Their opposing heuristics tend to prefer precisions about less relevant local variables.

The verification tasks in category `ECA` often contain only a few relevant variables, and in the majority of verification tasks all variables have the same domain type, and thus the heuristics Score-Min and Score-Max cannot always perform a meaningful selection here, and it degenerates to a heuristic about the number of distinct variables in the precision, hence, it does not have a positive impact in the category `ECA`. After all, the refinement-selection heuristics Score-Min and Score-Max are only applicable to verification tasks that contain program variables having different domain types. Note that the verification tasks in category `ECA` do not contain program variables that have loop-counter characteristics either, which leads us to believe that the refinement-selection heuristic Width-Min is superior here exactly due to the initial intention of the heuristic, namely, finding refinements that are associated with narrow precisions, so that tracking additional information is only needed in a smaller part of the state space.

As already pointed out before, the verification tasks of category `ProductLines` expose only very little non-determinism, and therefore, it is beneficial to start tracking information already from the initial program location, instead of refining deep in the

		BitVectorsReach	ControlFlow	DeviceDriversLinux64	ECA	Floats	Loops	ProductLines	Sequentialized	Simple	Overall
total		48	48	2 120	1 140	81	141	597	62	46	4 283
Length	Min	46	45	1 646	489	81	112	449	37	46	2 951
	Max	46	45	**1764**	507	80	115	361	31	46	2 995
Width	Min	46	45	1 661	508	81	112	469	39	46	3 007
	Max	46	45	1 746	481	80	114	357	35	46	2 950
Depth	Min	46	45	1 724	530	80	113	388	**42**	46	3 014
	Max	46	45	1 665	519	80	115	448	38	46	3 002
Score	Min	46	45	**1764**	534	81	114	414	37	46	3 081
	Max	46	45	1 665	394	80	115	364	29	46	2 784
Score & Width		46	45	1 741	561	81	112	417	39	46	**3088**
Width & Score		46	45	1 665	510	81	112	**472**	39	46	3 016
Random		46	45	1 687	529	81	112	381	39	46	2 966
None		46	45	1 661	**575**	80	114	453	**42**	46	3 062
Optimal		46	45	1 764	575	81	115	472	42	46	3 186
Oracle		46	45	1 809	631	81	115	502	47	46	3 322

Table 8.2: Number of solved verification tasks for the value analysis with refinement selection using different heuristics

rather complex state space of these verification tasks. This explains why the heuristics Length-Min and Depth-Min work especially well here, because these heuristics always leads to refinements that add information occurring near the initial program location, driving the verification towards shallow bugs without descending deep into the state space.

Finally, the compositional heuristics for refinement selection that we suggest do not bring any improvements for the predicate analysis, as they basically perform the same as the respective basic heuristics.

8.4.3 Refinement Selection for the Value Analysis

We now compare the different refinement-selection heuristics if used together with the value analysis.

In order to allow reproducibility of the evaluation, an example for a complete command line for performing guided refinement selection for the value analysis as

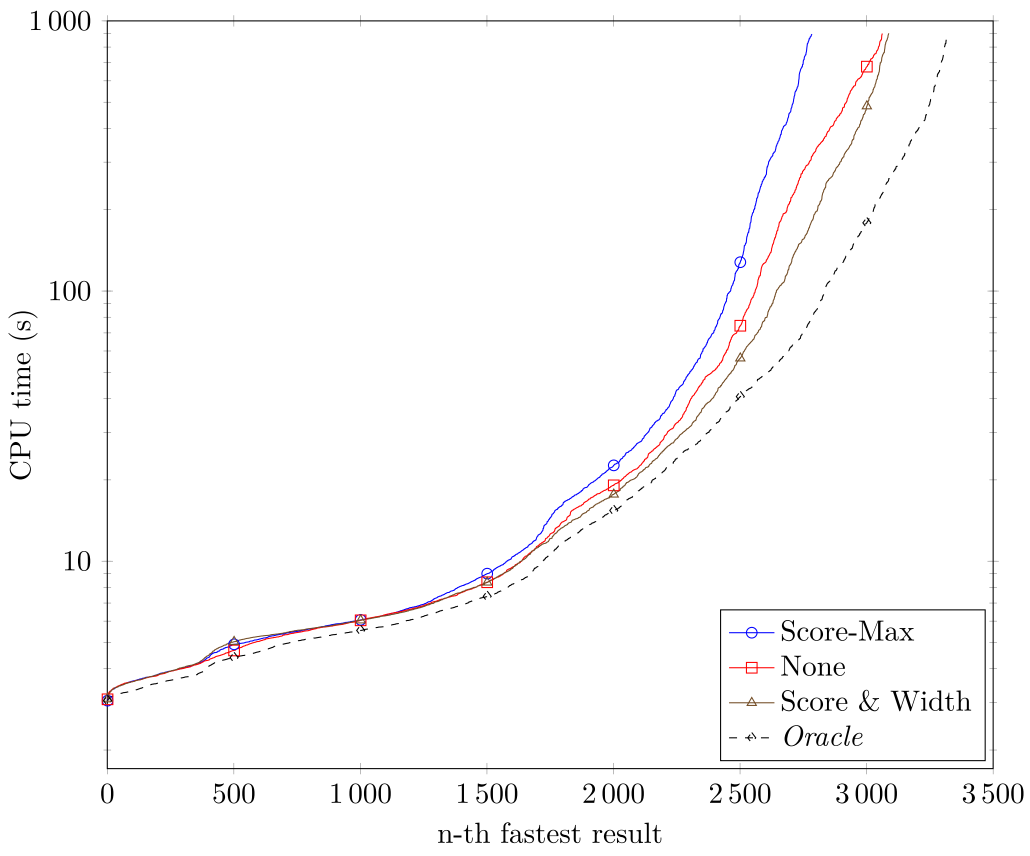


Figure 8.2: Quantile plot comparing the CPU time of value analysis without and with refinement selection using different heuristics

well as the full results and raw data are available on our supplementary web page [6].

The results are presented in Table 8.2, which is structured similarly to Table 8.1. First off, while guided refinement selection is quite effective for the value analysis, the configuration without explicit refinement selection performs quite well for the value analysis, as opposed to the predicate analysis, where it was clearly under-performing. This can be explained by the fact that the interpolation engine for the value analysis is implemented in CPAchecker itself and is thus designed and tuned specifically for software verification (cf. Chapter 5). In contrast, the predicate analysis uses an off-the-shelf SMT solver as interpolation engine, which is not designed specifically for software verification.

Similarly to the predicate analysis, none of the heuristics is the best for all classes of verification tasks. Again, the basic heuristic that works best over all verification tasks

[6] `http://www.sosy-lab.org/research/phd/loewe/#GuidedRefinementSelectionVa`

is Score-Min, which is especially well suited for the subset DeviceDriversLinux64 for the same reasons explained above. In fact, note that for the basic heuristics and categories of verification tasks presented in Tables 8.1 and 8.2, the number of verification tasks solved by the value analysis often correlates closely to the number of verification tasks solved by the predicate analysis, hence, for the respective categories we refer to the explanations given above for why a given refinement-selection heuristic may work well for a specific category. One difference between guided refinement selection for the value analysis and the predicate analysis lies in the performance of compositional refinement-selection heuristics, which in fact bring a slight improvement in case of the value analysis. Adding the refinement-selection heuristic Width-Min to the refinement-selection heuristic Score-Min allows the composition to just have the edge over the basic refinement-selection heuristic Score-Min in the category Overall, and joining the two refinement-selection heuristics the other way around allows the resulting composite refinement-selection heuristic Width & Score to edge out Width-Min in the category ProductLines.

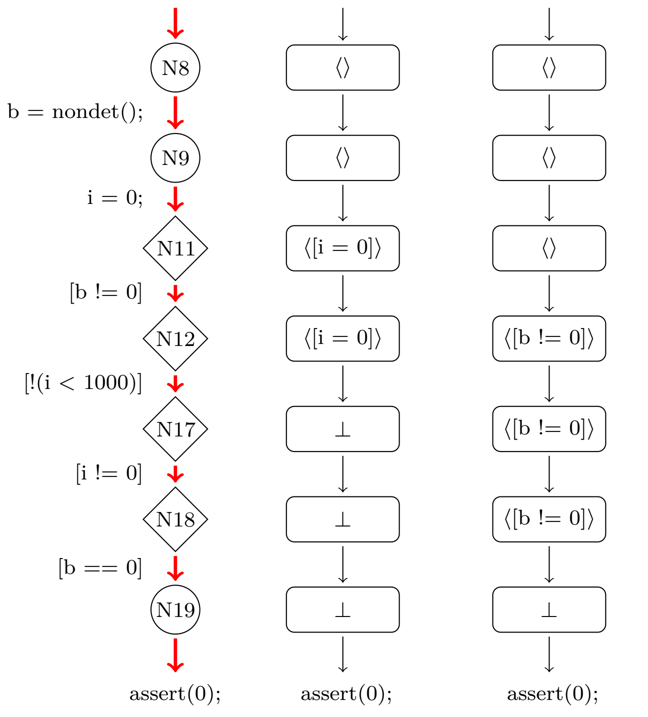
In conclusion, we must say that for the value analysis no basic or composite refinement-selection heuristics for itself leads to a greatly improved performance, however, when taking into account the results of the meta refinement-selection heuristics, especially those of heuristic *Oracle* (cf. Figure 8.2), then no one can deny the positive effect that guided refinement selection may have.

8.5 Refinement Selection for Composite Analyses

In Chapter 6 we discussed the potential of combining a value analysis and a predicate analysis to form a precise and efficient composite analysis based on CEGAR. There, the choice which domain is going to be refined is made statically, i. e., the supposedly cheaper value analysis is always preferred for a refinement.

For a given verification task this inflexible strategy may lead to an inefficient analysis, and we believe we can do better by bringing refinement selection to the next level, simply by making the decision whether to refine the value or the predicate analysis on-the-fly, right before a subsequent refinement. Figure 8.3 shows this via an example. For the given verification task, an analysis based on CEGAR, with an initially empty precision, will find the shown infeasible error path. The infeasibility of this path can be explained independently by both the valuations of the variables `int i` and `int b`, respectively, as shown by the two example interpolant sequences. As already pointed out before, it is generally advisable to track information about variables of boolean character, like the variable `int b`, rather than loop-counter variables, such as variable `int i`, because the latter may

```
#include <assert.h>

extern int nondet();
extern int f(int x);

int main() {

  int b = nondet();
  int i = 0;

  if (b != 0) {
    while (i < 1000) {
      f(i++);
    }
  }

  if (i != 0) {
    if (b == 0) {
      assert(0);
    }
  }
}
```



(a) example verification task (b) error path (c) bad sequence (d) good sequence

Figure 8.3: From left to right, (a) an example verification task, (b) an infeasible error path, and a (c) "bad" interpolant sequence and a (d) "good" interpolant sequence for this infeasible error path, the latter being not applicable to the value analysis, because it only contains inequalities

have far more valuations, and tracking loop counters would usually lead to expensive loop unrollings.

The composite analysis introduced in Chapter 6 always tries first to refine the value analysis and uses refinements of the predicate analysis only if necessary. Because the value analysis cannot track the constraint `[b != 0]`, the given error path of the verification task can —by the value analysis— only be excluded by tracking the loop-counter variable `int i`, which consequently would force unrolling the loop. If instead the predicate analysis could explicitly be chosen for refinement, then it could track the constraint `[b != 0]` for ruling out this infeasible error path. However, note that the predicate analysis could also start tracking predicates over the loop-counter variable `int i` and unroll the loop. Whether this would happen again depends solely on the internal heuristics of the interpolation engine being used.

Thus, for the error path in this example, we would like the verifier to refine using the predicate analysis, and we would like the interpolation engine to return the interpolant sequence shown on the right, and avoid interpolant sequences such as the

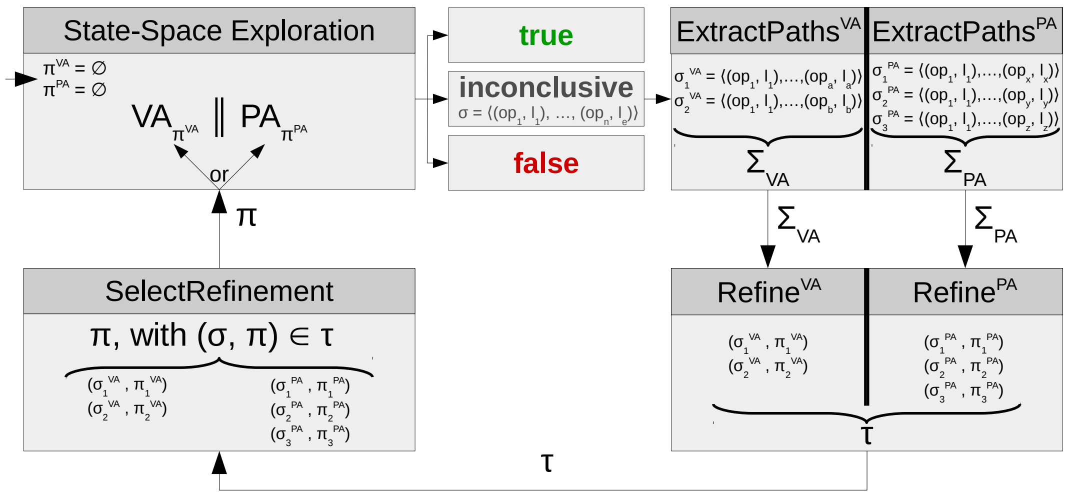

State-Space Exploration
$\pi^{VA} = \emptyset$
$\pi^{PA} = \emptyset$
$VA_{\pi^{VA}} \parallel PA_{\pi^{PA}}$
or
π
true
inconclusive
$\sigma = \langle(op_1, l_1), \ldots, (op_n, l_e)\rangle$
false
ExtractPaths^{VA}
ExtractPaths^{PA}
$\sigma_1^{VA} = \langle(op_1, l_1),\ldots,(op_a, l_a)\rangle$
$\sigma_2^{VA} = \langle(op_1, l_1),\ldots,(op_b, l_b)\rangle$
$\sigma_1^{PA} = \langle(op_1, l_1),\ldots,(op_x, l_x)\rangle$
$\sigma_2^{PA} = \langle(op_1, l_1),\ldots,(op_y, l_y)\rangle$
$\sigma_3^{PA} = \langle(op_1, l_1),\ldots,(op_z, l_z)\rangle$
Σ_{VA}
Σ_{PA}
SelectRefinement
π, with $(\sigma, \pi) \in \tau$
$(\sigma_1^{VA}, \pi_1^{VA})$
$(\sigma_2^{VA}, \pi_2^{VA})$
$(\sigma_1^{PA}, \pi_1^{PA})$
$(\sigma_2^{PA}, \pi_2^{PA})$
$(\sigma_3^{PA}, \pi_3^{PA})$
Refine^{VA}
Refine^{PA}
τ

Figure 8.4: Visualization of inter-analysis refinement selection for a composite analysis, here, consisting of a value analysis and a predicate analysis

one on the left, which references the loop counter `int i`.

Our evaluation in Chapter 6 already underlined the usefulness of combining different analyses, such as a value analysis and a predicate analysis, because different facts necessary to reason over a verification task can be handled by the analysis that can track a fact most efficiently. The refinement step is a natural place for choosing which of the analyses should track new information. Thus we extend the idea of refinement selection from an intra-analysis selection to an *inter*-analysis selection.

Mind that this approach is not specific to the value analysis and predicate analysis, but only requires two configurations of analyses that have support for CEGAR. Still, for the example depicted in Figure 8.4 we refer to a combination of a value analysis (VA) and a predicate analysis (PA). Our novel concept can be broken down into four distinct phases. The first phase is the standard exploration phase of CEGAR. The composite analysis performs the state-space exploration, constructing the abstract model using the initial, empty precision for all component analyses. In the figure, we refer to the precisions as π^{VA} and π^{PA} for the value analysis and the predicate analysis, respectively. If the outcome of the state-space exploration is either the verdict `true` or `false` then the analysis terminates. If the model contains an infeasible error path σ, then the model is inconclusive and, according to the CEGAR algorithm, a refinement is initiated.

With the refinement step, the second phase begins, which also marks the starting point of our novel approach for inter-analysis refinement selection. There, for all component analyses, we extract infeasible sliced prefixes stemming from the infeasible error path σ. Each program analysis provides its own strongest post-operator $\widehat{\mathsf{SP}}$, with each having different expressive power, and therefore, the set of infeasible sliced prefixes might differ among the component analyses. For example, with $\widehat{\mathsf{SP}}^{\mathsf{VA}}$ we can extract sliced prefixes that are infeasible due to contradictions involving non-linear arithmetic, while with $\widehat{\mathsf{SP}}^{\mathsf{PA}}$ we get sliced prefixes that are infeasible due to contradicting range predicates.

In the third phase, for each infeasible sliced prefix from the previous phase, a precision is computed by delegating to the default refinement routine Refine of the respective analysis. At the end of the third phase, the set τ contains the available refinements (as pairs of infeasible sliced prefixes and precisions) for all of the component analyses.

In the fourth phase, *one* suitable precision π (in the example, either π^{VA} or π^{PA}) is selected from the set τ, which is added to the respective precision of the component analysis for state-space exploration, finishing one CEGAR iteration. A proper strategy for inter-analysis refinement selection can be crucial for the verification effectiveness of the composite analysis, because, for an arbitrary verification task, there is no analysis superior to all other analyses, but one analysis may be a good fit for one class of verification tasks, but less suitable for another class, while it can be the other way around for a second analysis. Suppose, for example, an infeasible error path that can only by excluded by tracking that a certain variable is within some interval. Refining the value analysis would mean to enumerate all possible values of this variable, whereas the predicate analysis could track this efficiently using inequality predicates. The evaluation we present in the following provides evidence that inter-analysis refinement selection can be superior to statically preferring the refinement of a specific analysis, which is an improvement over our previous approach introduced in Chapter 6.

8.6 Evaluation of Inter-Analysis Refinement Selection

We now evaluate the effects of applying refinement selection to a composite analysis. To this end we compare the following three different analyses: (1) a sole predicate analysis without refinement selection, (2) a composite analysis of a value analysis and a predicate analysis (both without refinement selection), where refinements are always tried first with the value analysis and the predicate analysis is refined only if the value analysis cannot eliminate an infeasible error path (cf. Chapter 6), and (3) our novel composite analysis of a value analysis and a predicate analysis as defined in Section 8.5, where refinement selection is used within each domain and also to decide which domain to prefer in a refinement step.

	BitVectorsReach	ControlFlow	DeviceDriversLinux64	ECA	Floats	Loops	ProductLines	Sequentialized	Simple	Overall
total	48	48	2 120	1 140	81	141	597	62	46	4 283
PA	43	42	1 525	218	73	97	321	28	43	2 390
VA-PA-Composition	43	**45**	1 561	**484**	73	89	**390**	**38**	**46**	2 769
VA-PA-Composition–RefSel	**44**	**45**	**1652**	472	**74**	**98**	373	32	45	**2835**

Table 8.3: Number of solved verification tasks for the three different analyses

8.6.1 Configuration

We use the same experimental setup as in Section 8.4. For the configuration with inter-analysis refinement selection enabled, the predicate analysis is favoured for performing a refinement in those cases where the value analysis only has infeasible sliced prefixes available that reference loop-counter variables. In all other cases, the value analysis is favoured for performing a refinement.

In order to allow reproducibility of the evaluation, an example for a complete command line for performing inter-analysis refinement selection for the composite analysis as well as the full results and raw data are available on our supplementary web page[7].

8.6.2 Results

Table 8.3 shows the results for this comparison. Confirming the previous results from Chapter 6, a composition of value analysis and predicate analysis without refinement selection (row VA-PA-Composition) is already more effective than the predicate analysis alone (row PA). However, the composite approach also has a weak spot, as it fails more often in category Loops due to state-space explosion, whereas the predicate analysis alone succeeds. The third configuration (row VA-PA-Composition–RefSel) takes the idea of refinement selection to the next level. While in the former composite approach the value analysis is always refined first, and the predicate analysis only if the value analysis cannot eliminate an infeasible error path, our novel composite analysis uses inter-analysis refinement selection to decide whether a refinement for the value or for the predicate analysis is thought to be more effective. Consider, for example, the results in category Loops. In this category the plain predicate

[7]`http://www.sosy-lab.org/research/phd/loewe/#InterAnalysisRefinementSelection`

analysis (row PA) out-performs the naive composition of the value analysis and the predicate analysis (row VA-PA-Composition). If, however, we apply inter-analysis refinement selection to decide which analysis to refine for a given infeasible error path, as done by our novel approach (row VA-PA-Composition–RefSel), then this shows a higher verification effectiveness than the predicate analysis for verification tasks where reasoning about loops is essential, i. e., in category LOOPS, and it clearly out-performs the plain predicate analysis as well as the composition of the value analysis and the predicate analysis over all verification tasks.

8.7 Conclusion

In this chapter we introduced the notion of *guided* refinement selection, a method that is able to guide the construction of an abstract model in a direction that is thought to be beneficial for the effectiveness and efficiency of the verification process. In a thorough evaluation incorporating several refinement-selection heuristics we underlined the great potential of refinement selection for the value analysis, for the predicate analysis, as well as for a composite analysis of a value and a predicate analysis for which we also introduced and validated the concept of *inter-analysis* refinement selection. This opens a fundamentally new view on verification of models with different characteristics. Instead of using portfolio checking, or trying several different abstract domains, we can, in one single tool, fully automatically self-configure the verifier, according to the property to be verified and the abstract domain that can best analyze the paths that are encountered during the analysis.

8.7.1 Lessons Learned

The most interesting insight from this chapter is that an analysis may become far more efficient if guided refinement selection is applied and not simply rely on the interpolants provided through the standard heuristics implemented in off-the-shelf interpolation engines, as apparently their internal heuristics are not always well suited for software verification. The heuristics we presented allow the extraction of *custom* interpolants, and the success of our heuristics underlines that guided refinement selection matters.

8.7.2 Challenge

While the results of the meta-heuristic *Oracle* are highly encouraging, it also means that we are still missing a single heuristic for guided refinement selection that is superior in a wider range of verification tasks, e. g., over multiple categories of SV-COMP. More research is needed in this area, and therefore, it would be interesting to investigate heuristics that, for example, use dynamic information from the analysis. There, instead of penalizing a loop-counter variable according to its domain type, one could delay the penalty until a certain threshold is reached on the number of values for this variable, similar to dynamic precision adjustment [22]. Especially for the predicate analysis, it would be interesting to investigate heuristics that not only look at the domain type, but also how the variables are referenced in the precision, e. g., an equality predicate for a loop counter usually leads to unrollings of loops, while an inequality might avoid the unrolling of a loop. Similarly, for inter-analysis refinement selection more advanced selection strategies could be defined that always allow to systematically select the most appropriate domain for performing a subsequent refinement.

8.7.3 Proposition

Ideas and actual solutions to tackle the challenges mentioned above would be highly valuable, however, these are not regarded as being in the scope of this thesis. Instead, we accepted a different challenge and participated in SV-COMP'16, with a version of CPAchecker incorporating many concepts described throughout this thesis.

9 Contribution to SV-COMP'16

So far, we contributed at least one verifier to every edition of SV-COMP, and the plain value analysis (cf. Section 3) and the value analysis based on CEGAR and interpolation (cf. Section 4) helped the CPAchecker team to win numerous gold, silver and bronze medals in SV-COMP over the years. Furthermore, the composite analysis featuring the value analysis and the predicate analysis (cf. Section 6) won four silver medals in SV-COMP'13, one of which was awarded for its 2nd place in the overall ranking [11, 83].

9.1 Configuration

For SV-COMP'16 we contributed the verifier CPA-RefSel [84], which is based on various ideas and concepts described in this thesis. Similar to our contribution to SV-COMP'13 it again features a composite analysis of the value analysis and the predicate analysis, but getting a major boost from applying intra- and inter-analysis refinement selection, as described in the previous chapter. In detail, both the value analysis and the predicate analysis perform guided refinement selection, and are configured to favor refinements with a low domain-type score, as this works particularly well for many verification tasks from SV-COMP'16. In order to make refinement selection to work best for the predicate analysis, we configured it to perform abstraction computations at loop heads or whenever control flow joins (cf. Section 7.6.3). Furthermore, we configured the component for inter-analysis refinement selection in such a way, that always the one domain is refined which has the lower domain type associated for a refinement.

Beyond incorporating refinement selection, we also added a few smaller optimizations to our verifier. For example, we augmented the predicate analysis with the concept of a scoped precision, in a similar fashion as introduced for the value analysis before (cf. Section 5.3). With this, predicates about a program variable are now not only added at program locations dictated by the interpolants, but at all program locations that are in the scope of the respective program variable, i. e., predicates about local variables are added to all program locations of the respective function, while predicates about global variables are added to all program locations of the verification task.

Furthermore, to accommodate the better performance of the plain value analysis over the value analysis with CEGAR and interpolation for verification tasks exposing a high level of determinism (cf. Section 5.4.4), we added a feature such that abstraction computations for the value analysis are only performed once the level of non-determinism reaches a certain, configurable threshold. Thus, the value analysis with CEGAR and interpolation behaves exactly like the plain value analysis for verification tasks that expose only very little non-determinism.

Together, all these concepts allow our verifier CPA-RefSel to be highly competitive for control-flow-heavy verification tasks that encode reachability problems, as witnessed by the good results CPA-RefSel obtained in SV-COMP'16, on which we detail and extend in the following section.

9.2 Results of SV-COMP'16 and beyond

As already shown in the previous chapter, especially the addition of both intra- and inter-analysis refinement selection allows for much improved results in the category DeviceDriversLinux64 (cf. Table 8.3). We claim this to be the prime reason why our verifier CPA-RefSel was so successful there, winning the *gold medal* by quite some margin and taking the crown from the verifier Blast, which has been tuned for this category over the last years and dominated there so far in SV-COMP.

In Figure 9.1 we present an overview of the correctly solved verification tasks from category DeviceDriversLinux64 for a selection of verifiers that participated in SV-COMP'16. For the plot, we used the data provided by the organizers of SV-COMP [1], and we compare the results of our verifier CPA-RefSel to the verifiers UAutomizer, CPA-Seq, and SMACK+Corral which achieved the 1st, 2nd, and 3rd place, respectively, in category Overall in SV-COMP'16. CPA-Seq also got the 2nd place in DeviceDriversLinux64. In this comparison our verifier CPA-RefSel comes out as clear winner, as it solves considerably more verification tasks correctly than any other verifier from SV-COMP'16.

Since SV-COMP'16, we further improved and tuned our approach on a purely technical level. For example, we further optimized the refinement-selection heuristics, as well as the strategy for enabling abstraction computations for the value analysis in presence of non-determinism, and the predicate analysis no longer runs in bit-precise mode as in SV-COMP'16. Other than for SV-COMP'16, all concepts are by now fully integrated into the trunk version of CPAchecker, and based on this implementation we performed a re-evaluation.

[1] The data is available from `http://sv-comp.sosy-lab.org/2016/results/results-verified/`

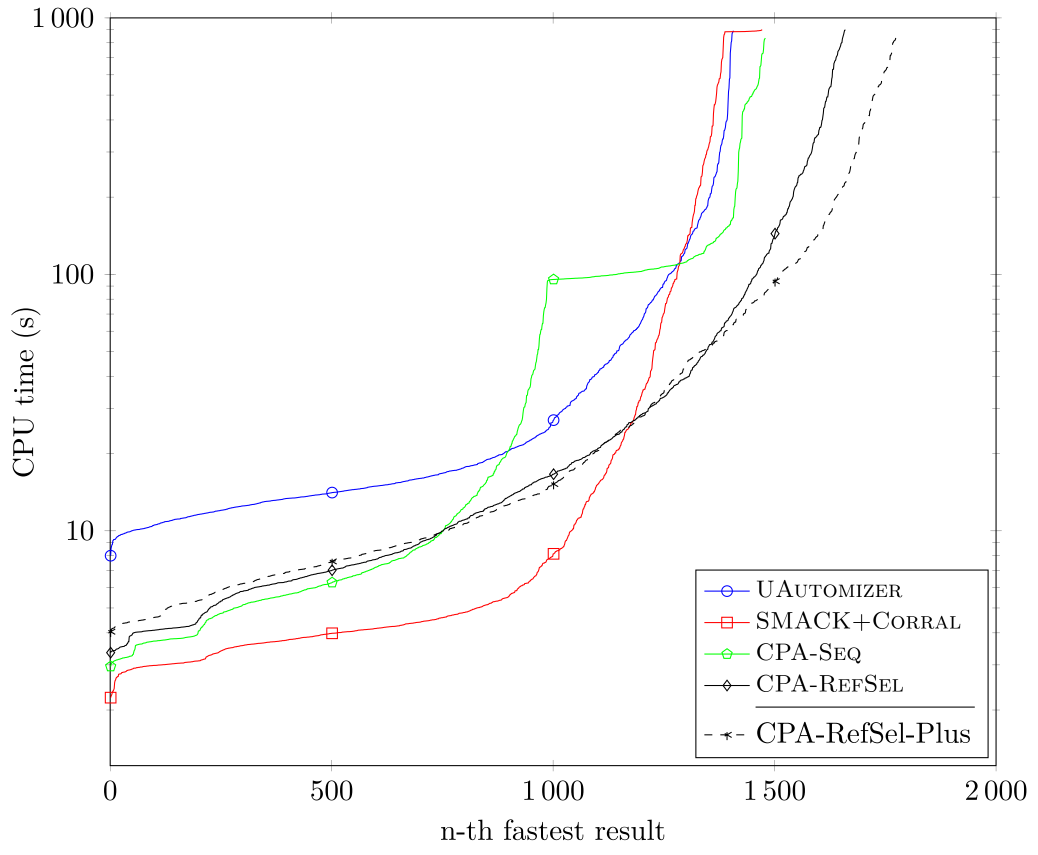


Figure 9.1: Quantile plot comparing for several verifiers the number of correctly solved verification tasks for category `DeviceDriversLinux64`

For this re-evaluation [2], we once more replicated the same experimental setup as used in SV-COMP'16, again tasking BenchExec to enforce the same resource limitations as defined for SV-COMP'16. In Figure 9.1 we also show the results of running a configuration of CPAchecker similar to CPA-RefSel, which we refer to as CPA-RefSel-Plus, and which is built using the CPAchecker trunk revision 21 270 from May the 15th, 2016.

In order to allow reproducibility of the evaluation, an example for a complete command line for running CPA-RefSel-Plus as well as the full results and raw data are available on our supplementary web page [3].

From the graph one can observe that CPA-RefSel-Plus solves even more verification tasks correctly than CPA-RefSel did in the identical evaluation environment of

[2] Again, we do not perform any witness checking in this evaluation, so the results obtained here differ slightly from the official results of SV-COMP'16.

[3] `http://www.sosy-lab.org/research/phd/loewe/#CpaRefSelPlus`

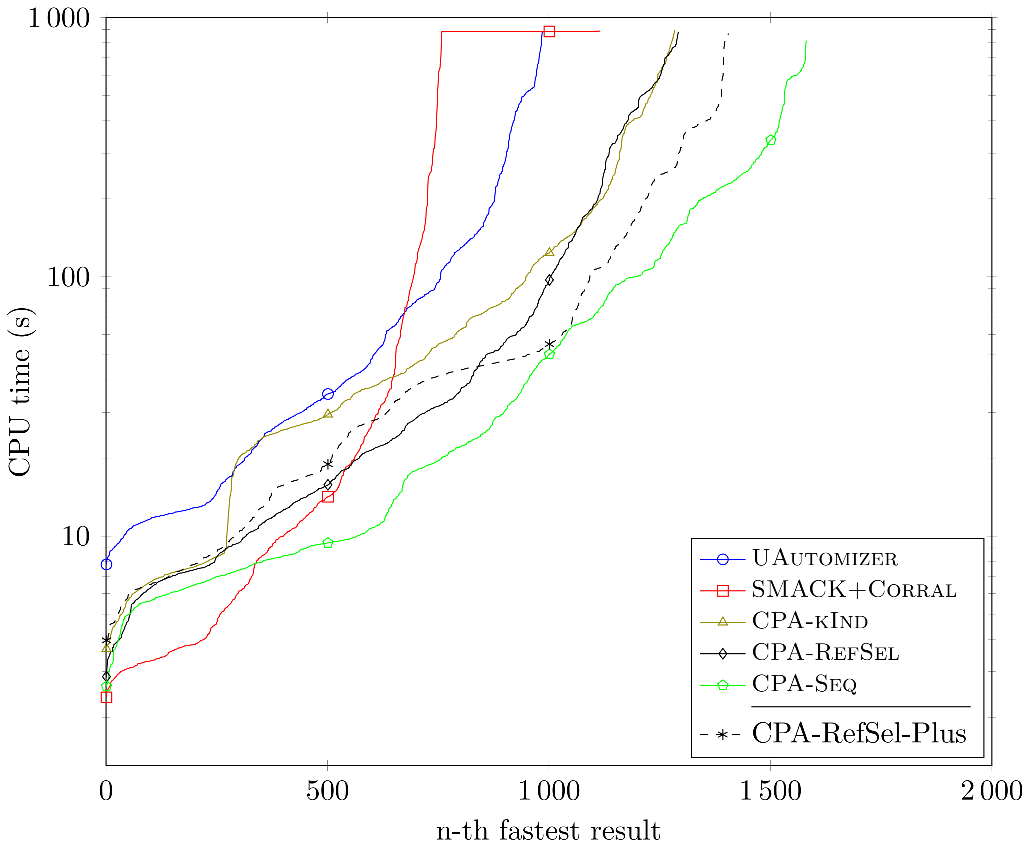


Figure 9.2: Quantile plot comparing for several verifiers the number of correctly solved verification tasks from category `IntegersControlFlow`

SV-COMP'16. There, CPA-RefSel solved 1 660 verification tasks correctly, while the runner-up CPA-Seq solved only 1 479 verification tasks correctly. CPA-RefSel-Plus is now able to solve 1 774 out of the 2 120 verification tasks from the category `DeviceDriversLinux64` correctly. Mind that, when comparing the scores using the official scoring schema of SV-COMP'16, then CPA-RefSel-Plus obtains a score of 3 138, which clearly beats the 2 822 points of CPA-Seq, which was the best verifier after CPA-RefSel, the latter achieving a best of 3 191 points.

The only other category for which our verifier and the contributions in this thesis are designed for is the category `IntegersControlFlow`, excluding the 98 verification tasks from the subset `Recursive`[4]. This set contains 2 233 verification tasks and represents the union over the complete categories `ControlFlow`, `ECA`, `Loops`, `ProductLines`, `Sequentialized`, and `Simple` as introduced above in this thesis (cf. Table 3.1).

[4]None of the contributions in this thesis are targeted at recursion.

We compare the same verifiers over the same criteria as before, but add the verifier CPA-kInd which won the silver medal in category **IntegersControlFlow** in SV-COMP'16. From the overview in Figure 9.2 one can see that CPA-RefSel just out-performs CPA-kInd in terms of correctly solved verification tasks. The portfolio-like approach of CPA-Seq, which combines up to five different verification techniques, is best for this quite diverse set of different verification tasks which spans over six sub-categories. While our verifier CPA-RefSel-Plus represents a significant improvement over CPA-RefSel of SV-COMP'16, it is not able to close the gap to CPA-Seq.

However, we firmly believe that even better results are possible if, for an *individual* verification task, the most fitting heuristic for intra- and inter-analysis refinement selection could be applied (cf. Section 8.4), and with the availability of the implementation in the CPAchecker framework, it is easily doable to integrate and extend these approaches for the submission of the CPAchecker team to SV-COMP'17.

10 Summary and Future Research

In the final chapter we provide a summary of this thesis and we give a brief outlook on further research directions that build on the presented concepts and ideas.

10.1 Summary

In this thesis we presented several concepts that may help to pave the way such that automatic software verification becomes more relevant in practice. First, in Chapter 1, we established that there is actual demand for automatic software verification in practice, and we introduced the theoretical background needed for properly describing our approaches in Chapter 2.

In our first contribution, described in Chapter 3, we introduced the value analysis, i. e., an analysis that, other than the pre-dominant symbolic analyses, tracks the concrete valuations of program variables in a verification task. This design decision allows a highly efficient successor computation, and, as our evaluation showed, this leads to an efficient overall analysis in many cases, also for verification tasks where symbolic approaches are less well suited. However, this value analysis also comes with the disadvantages of being prone to state-space explosion and being imprecise in the presence of non-determinism.

Mitigating the effects of state-space explosion for the value analysis was the subject of Chapter 4, where we proposed to extend the value analysis by the CEGAR technique. To this end we defined, designed and built from the ground up (1) the precision, (2) the feasibility check, and (3) the refinement procedure with a novel value-interpolation technique for the value domain, such that together with the state-space exploration algorithm of the value analysis the construction of a value analysis with CEGAR and interpolation became reality. The evaluation of the value analysis with CEGAR and interpolation revealed that our novel approach allows the successful verification of many verification tasks that the plain value analysis cannot solve, but also, that the plain value analysis is better suited for other verification tasks, and in total still solves more verification tasks than the value analysis with CEGAR and interpolation.

Consequently, in Chapter 5 we continued with several optimizations and we detailed how we tuned both the value-interpolation component as well as the overall refinement process of the value analysis. For the interpolation component we devised three heuristics, namely (1) to perform interpolation only over the deepest infeasible suffix, (2) skip interpolation in cases subsequent candidate interpolants are equal, and (3) skip interpolation in cases subsequent candidate interpolants are equivalent, with the added bonus that all three heuristics are compatible with each other. Applying all three heuristics at once revealed that the number of interpolation queries are reduced to a third when compared to the case where none of the heuristics are used. In order to optimize the overall refinement process, we chose to effectively disable lazy abstraction, by installing a scoped precision and restarting the state-space exploration after each refinement. As our evaluation showed, this allows for a far more efficient verification process, as the number of refinements needed for most verification tasks is greatly reduced.

Furthermore, the level of non-determinism of a verification task was identified as a valid indicator whether to better disable or enable CEGAR for the value analysis. Finally, we discussed the applicability of the refinement for the value domain to other domains as well, e. g., the octagon domain or symbolic execution, and how it facilitates regression verification.

The result of Chapter 5 marks a milestone in our work, as we then had a novel analysis available that supports CEGAR with interpolation, and so, from this point on, we were able to formulate any idea for CEGAR or interpolation for the value analysis, as well as for existing analyses based on CEGAR and interpolation, such as, e. g., a predicate analysis.

Accordingly, in the subsequent chapters, we focused on techniques that lead to advancements in both the value analysis and the predicate analysis, with the logical first step of combining the two analyses to obtain a precise and efficient composite analysis, that we described in Chapter 6. The goal of this composite analysis was —depending from the standpoint— to either make the value analysis more precise, or to make the predicate analysis more efficient for specific classes of verification tasks. We achieved this goal, as witnessed by our evaluation as well as by the positive results we obtained in SV-COMP'13, where this novel approach secured us the 2$^{\text{nd}}$ place in the overall ranking besides three more silver medals in other categories.

In Chapter 7 we continued with exploring techniques that may lead to advancements in analyses based on CEGAR, and we argued that an interpolation engine should be considered as black-box which not always is well suited for software verification. This is because it gets as input some interpolation query and returns interpolants according to its internal heuristics, which are usually not tuned specifically for software verification. Due to that, a software verifier does not have much control on which interpolants are returned, and so, it often happens that the returned interpolants are not well suited for the further verification process. To this end, we proposed the concept of infeasible sliced prefixes. With this concept, one is able to extract from a single infeasible error path a set of infeasible sliced prefixes, where each of those infeasible sliced prefixes can be used to exclude the original infeasible error path from future CEGAR iterations. We developed a general algorithm for extracting infeasible sliced prefixes and evaluated it both for the value and the predicate analysis, and with the availability of, in general, multiple infeasible sliced prefixes, the refinement component is free to choose any of the infeasible sliced prefixes for a subsequent refinement. The evaluation we performed confirmed for both the value and the predicate analysis that selecting different infeasible sliced prefixes may have a major influence on the verification effectiveness of the overall analysis.

In Chapter 8 we extended this concept, as we devised various heuristics that are geared to systematically select those infeasible sliced prefixes that lead to suitable refinements, forming the technique we refer to as guided refinement selection. In another large-scale evaluation we compare these heuristics for guided refinement selection against each other, with the results that (1) the default approach without guided refinement selection is hardly ever the best, (2) selecting refinements with a low domain-type score works particularly well, but also, (3) that there is no clear winner over all different refinement-selection heuristics, because each refinement-selection heuristic works well in one class of verification tasks, but is less suited for another class of verification tasks. Besides intra-refinement selection, we also presented the concept of inter-analysis refinement selection, where the refinement component of a composite analysis decides which of its component analyses is best to be refined in order to exclude a specific infeasible counterexample. This technique marks a major improvement over the previous approach for the composite analysis, where always a pre-defined analysis was refined first.

Finally, in Chapter 9 we described how we plugged ideas and concepts of this thesis together to build the software verifier CPA-RefSel, which won the gold medal in category DeviceDriversLinux64 in SV-COMP'16, and we showed how we improved our verifier to be also on par with the world's leading software verifiers in the category IntegersControlFlow.

10.2 Future Research

Despite the fact that we regard this thesis as self-contained, we would like to point the reader to some possible directions for future research.

The plain value analysis (cf. Chapter 3) could benefit from mere technical improvements, like support for IEEE-754 floating-point arithmetic. Reasoning about pointers, arrays or data structures on the heap could be added by better interfacing to a CPA tracking concrete or symbolic memory graphs [53, 85].

The technique we refer to value interpolation (cf. Chapter 4) was already applied successfully to the octagon domain and to symbolic execution, and it would be interesting to evaluate its applicability to the aforementioned symbolic memory graphs. Combining these techniques would for example allow to abstract the exact shape of a linked list in a scenario where this information is not relevant, but in another case, the analysis could track the elements of the linked list up to a certain length such that an infeasible error path can be excluded. Also, adding the value interpolation and CEGAR as devised in this thesis to an analysis based on binary decision diagrams [32, 36] would be worth investigating.

The composite analysis presented in this thesis had great success (cf. Chapter 6), mainly because it combines two fundamentally different verification approaches. With CPAchecker it is easy to mix and match various verification approaches, and novel combinations of existing or new CPAs sure would yield benefits for special use cases. Furthermore, new insights here may also have an impact on (inter-analysis) refinement selection (cf. Chapters 8), i. e., where inter-analysis refinement selection has to decide which analysis is better to be refined. In that regard, it would also be possible to evaluate inter-analysis refinement selection with more than just two different CPAs. However, before investigating that, better heuristics for intra- and inter-analysis refinement selection would be needed. For example, so far, we only investigated which program variables are referenced by different refinements, but for the predicate analysis, one could also differentiate refinements based on whether more equality predicates or more inequality predicates are being referenced. The former are generally considered less suitable, because they may lead to loop unrollings. Symbolic execution is being used in many fields, especially for testing, and refinement selection helped to boost the performance of the analysis for symbolic execution in CPAchecker significantly [25]. Hence, it would be interesting if the successful combination of symbolic execution and refinement selection has an impact on more applications of symbolic execution. Additionally, refinement selection and infeasible sliced prefixes could be used for finding and generating invariants, and make them available to other analyses [17]. Furthermore, being able to extract more infeasible sliced prefixes while also applying ABE would perhaps allow guided refinement selection to work even better for the predicate analysis. Finally, we firmly believe that guided refinement selection could be even more successful if it could make use of domain knowledge, e. g., knowing before starting the verification process which program variables are likely to be needed for reasoning over a given verification task. We are proud that the value analysis with CEGAR and the composite analysis of the value analysis and the predicate analysis are actively used by practitioners and researchers as part of the Linux Driver Verification program [1], but having more industrial partners would allow us to obtain more insights and tune the concepts and ideas, as well as the whole analyses themselves, in order to enable automatic software verification in practice.

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

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Craig Interpretation. In Antoine Miné and David Schmidt, editors, *Proceedings of SAS, Deauville, France*, volume 7460 of *LNCS*, pages 300–316. Springer, 2012.

[3] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. From Under-Approximations to Over-Approximations and Back. In Cormac Flanagan and Barbara König, editors, *Proceedings of TACAS, Tallinn, Estonia*, volume 7214 of *LNCS*, pages 157–172. Springer, 2012.

[4] Aws Albarghouthi, Arie Gurfinkel, Yi Li, Sagar Chaki, and Marsha Chechik. UFO: Verification with Interpolants and Abstract Interpretation (Competition Contribution). In Nir Piterman and Scott A. Smolka, editors, *Proceedings of TACAS, Rome, Italy*, volume 7795 of *LNCS*, pages 637–640. Springer, 2013.

[5] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. An Extension of Lazy Abstraction with Interpolation for Programs with Arrays. *Formal Methods in System Design*, 45(1):63–109, 2014.

[6] Sven Apel, Dirk Beyer, Karlheinz Friedberger, Franco Raimondi, and Alexander von Rhein. Domain Types: Abstract-Domain Selection Based on Variable Usage. In Valeria Bertacco and Axel Legay, editors, *Proceedings of HVC, Haifa, Israel*, volume 8244 of *LNCS*, pages 262–278. Springer, 2013.

[7] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In John Launchbury and John C. Mitchell, editors, *Proceedings of POPL, Portland, OR, USA*, pages 1–3. ACM, 2002.

[8] Jiri Barnat, Lubos Brim, Vojtech Havel, Jan Havlícek, Jan Kriho, Milan Lenco, Petr Rockai, Vladimír Still, and Jirí Weiser. DiVinE 3.0 — An Explicit-State Model Checker for Multithreaded C & C++ Programs. In Natasha Sharygina and Helmut Veith, editors, *Proceedings of CAV, Saint Petersburg, Russia*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013.

[9] Gérard Basler, Alastair F. Donaldson, Alexander Kaiser, Daniel Kroening, Michael Tautschnig, and Thomas Wahl. SatAbs: A Bit-Precise Verifier for C Programs (Competition Contribution). In Cormac Flanagan and Barbara König, editors, *Proceedings of TACAS, Tallinn, Estonia*, volume 7214 of *LNCS*, pages 552–555. Springer, 2012.

[10] Dirk Beyer. Competition on Software Verification (SV-COMP). In Cormac Flanagan and Barbara König, editors, *Proceedings of TACAS, Tallinn, Estonia*, volume 7214 of *LNCS*, pages 504–524. Springer, 2012.

[11] Dirk Beyer. Second Competition on Software Verification (Summary of SV-COMP 2013). In Nir Piterman and Scott A. Smolka, editors, *Proceedings of TACAS, Rome, Italy*, volume 7795 of *LNCS*, pages 594–609. Springer, 2013.

[12] Dirk Beyer. Status Report on Software Verification (Competition Summary SV-COMP 2014). In Erika Ábrahám and Klaus Havelund, editors, *Proceedings of TACAS, Grenoble, France*, volume 8413 of *LNCS*, pages 373–388. Springer, 2014.

[13] Dirk Beyer. Software Verification and Verifiable Witnesses (Report on SV-COMP 2015). In Christel Baier and Cesare Tinelli, editors, *Proceedings of TACAS, London, UK*, volume 9035 of *LNCS*, pages 401–416. Springer, 2015.

[14] Dirk Beyer. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). In Marsha Chechik and Jean-François Raskin, editors, *Proceedings of TACAS, Eindhoven, The Netherlands*, volume 9636 of *LNCS*, pages 887–904. Springer, 2016.

[15] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. Software Model Checking via Large-Block Encoding. In Armin Biere and Carl Pixley, editors, *Proceedings of FMCAD, Austin, TX, USA*, pages 25–32. IEEE, 2009.

[16] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, and Andreas Stahlbauer. Witness Validation and Stepwise Testification across Software Verifiers. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of FSE, Bergamo, Italy*, pages 721–733. ACM, 2015.

[17] Dirk Beyer, Matthias Dangl, and Philipp Wendler. Boosting k-Induction with Continuously-Refined Invariants. In Daniel Kroening and Corina S. Pasareanu, editors, *Proceedings of CAV, San Francisco, CA, USA*, volume 9206 of *LNCS*, pages 622–640. Springer, 2015.

[18] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The Software Model Checker BLAST. *Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

[19] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. Conditional Model Checking: A Technique to Pass Information between Verifiers. In Will Tracz, Martin P. Robillard, and Tevfik Bultan, editors, *Proceedings of FSE, Cary, NC, USA*, page 57. ACM, 2012.

[20] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path Invariants. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of PLDI, San Diego, CA, USA*, pages 300–309. ACM, 2007.

[21] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In Werner Damm and Holger Hermanns, editors, *Proceedings of CAV, Berlin, Germany*, volume 4590 of *LNCS*, pages 504–518. Springer, 2007.

[22] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Program Analysis with Dynamic Precision Adjustment. In *Proceedings of ASE, L'Aquila, Italy*, pages 29–38. IEEE, 2008.

[23] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of CAV, Snowbird, UT, USA*, volume 6806 of *LNCS*, pages 184–190. Springer, 2011.

[24] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. Predicate Abstraction with Adjustable-Block Encoding. In Roderick Bloem and Natasha Sharygina, editors, *Proceedings of FMCAD, Lugano, Switzerland*, pages 189–197. IEEE, 2010.

[25] Dirk Beyer and Thomas Lemberger. Symbolic Execution with CEGAR. In *Proceedings of ISoLA, Corfu, Greece*. Springer, 2016.

[26] Dirk Beyer and Stefan Löwe. Explicit-State Software Model Checking Based on CEGAR and Interpolation. In Vittorio Cortellessa and Dániel Varró, editors, *Proceedings of FASE, Rome, Italy*, volume 7793 of *LNCS*, pages 146–162. Springer, 2013.

[27] Dirk Beyer, Stefan Löwe, Evgeny Novikov, Andreas Stahlbauer, and Philipp Wendler. Precision Reuse for Efficient Regression Verification. In Bertrand

Meyer, Luciano Baresi, and Mira Mezini, editors, *Proceedings of FSE, Saint Petersburg, Russian Federation*, pages 389–399. ACM, 2013.

[28] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Benchmarking and Resource Measurement. In Bernd Fischer and Jaco Geldenhuys, editors, *Proceedings of SPIN, Stellenbosch, South Africa*, volume 9232 of *LNCS*, pages 160–178. Springer, 2015.

[29] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Refinement Selection. In Bernd Fischer and Jaco Geldenhuys, editors, *Proceedings of SPIN, Stellenbosch, South Africa*, volume 9232 of *LNCS*, pages 20–38. Springer, 2015.

[30] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Sliced Path Prefixes: An Effective Method to Enable Refinement Selection. In Susanne Graf and Mahesh Viswanathan, editors, *Proceedings of FORTE, Grenoble, France*, volume 9039 of *LNCS*, pages 228–243. Springer, 2015.

[31] Dirk Beyer and Alexander K. Petrenko. Linux Driver Verification (Position Paper). In Tiziana Margaria and Bernhard Steffen, editors, *Proceedings of ISoLA, Heraklion, Greece*, volume 7610 of *LNCS*, pages 1–6. Springer, 2012.

[32] Dirk Beyer and Andreas Stahlbauer. BDD-Based Software Model Checking with CPAchecker. In Antonín Kucera, Thomas A. Henzinger, Jaroslav Nesetril, Tomás Vojnar, and David Antos, editors, *Proceedings of MEMICS, Znojmo, Czech Republic*, volume 7721 of *LNCS*, pages 1–11. Springer, 2012.

[33] Dirk Beyer and Philipp Wendler. Algorithms for Software Model Checking: Predicate Abstraction vs. Impact. In Gianpiero Cabodi and Satnam Singh, editors, *Proceedings of FMCAD, Cambridge, UK*, pages 106–113. IEEE, 2012.

[34] Johannes Birgmeier, Aaron R. Bradley, and Georg Weissenbacher. Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR). In Armin Biere and Roderick Bloem, editors, *Proceedings of CAV, Vienna, Austria*, volume 8559 of *LNCS*, pages 831–848. Springer, 2014.

[35] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A Static Analyzer for Large Safety-Critical Software. In Ron Cytron and Rajiv Gupta, editors, *Proceedings of PLDI, San Diego, CA, USA*, pages 196–207. ACM, 2003.

[36] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[37] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Richard Draves and Robbert van Renesse, editors, *Proceedings of OSDI, San Diego, CA, USA*, pages 209–224. USENIX Association, 2008.

[38] Géraud Canet, Pascal Cuoq, and Benjamin Monate. A Value Analysis for C Programs. In *Proceedings of SCAM, Edmonton, AB, Canada*, pages 123–124. IEEE, 2009.

[39] Roberto Cavada, Alessandro Cimatti, Anders Franzén, Krishnamani Kalyanasundaram, Marco Roveri, and R. K. Shyamasundar. Computing Predicate Abstractions by Integrating BDDs and SMT Solvers. In Jason Baumgartner and Mary Sheeran, editors, *Proceedings of FMCAD, Austin, TX, USA*, pages 69–76. IEEE, 2007.

[40] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An Interpolating SMT Solver. In Alastair F. Donaldson and David Parker, editors, *Proceedings of SPIN, Oxford, UK*, volume 7385 of *LNCS*, pages 248–254. Springer, 2012.

[41] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott A. Smolka, editors, *Proceedings of TACAS, Rome, Italy*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.

[42] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. A Simple and Flexible Way of Computing Small Unsatisfiable Cores in SAT Modulo Theories. In João Marques-Silva and Karem A. Sakallah, editors, *Proceedings of SAT, Lisbon, Portugal*, volume 4501 of *LNCS*, pages 334–339. Springer, 2007.

[43] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM*, 50(5):752–794, 2003.

[44] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2001.

[45] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, *Proceedings of ICSE, Limerick, Ireland*, pages 439–448. ACM, 2000.

[46] Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Programs. In *Proceedings of ISOP, Paris, France*, pages 106–130. Dunod, 1976.

[47] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Proceedings of POPL, Los Angeles, CA, USA*, pages 238–252. ACM, 1977.

[48] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Proceedings of POPL, Tucson, AZ, USA*, pages 84–96. ACM Press, 1978.

[49] William Craig. Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.

[50] Yulia Demyanova, Helmut Veith, and Florian Zuleger. On the Concept of Variable Roles and its Use in Software Analysis. In Barbara Jobstman and Sandip Ray, editors, *Proceedings of FMCAD, Portland, OR, USA*, pages 226–230. IEEE, 2013.

[51] Edsger W. Dijkstra. Notes on Structured Programming. circulated privately, 1970.

[52] Vijay D'Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant Strength. In Gilles Barthe and Manuel V. Hermenegildo, editors, *Proceedings of VMCAI, Madrid, Spain*, volume 5944 of *LNCS*, pages 129–145. Springer, 2010.

[53] Kamil Dudka, Petr Peringer, and Tomás Vojnar. Byte-Precise Verification of Low-Level List Manipulation. In Francesco Logozzo and Manuel Fähndrich, editors, *Proceedings of SAS, Seattle, WA, USA*, volume 7935 of *LNCS*, pages 215–237. Springer, 2013.

[54] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[55] Stephan Falke, Florian Merz, and Carsten Sinz. LLBMC: Improved Bounded Model Checking of C Programs Using LLVM (Competition Contribution). In Nir Piterman and Scott A. Smolka, editors, *Proceedings of TACAS, Grenoble, France*, volume 7795 of *LNCS*, pages 623–626. Springer, 2013.

[56] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[57] Jeffrey Fischer, Ranjit Jhala, and Rupak Majumdar. Joining Dataflow with Predicates. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of FSE, Lisbon, Portugal*, pages 227–236. ACM, 2005.

[58] Robert W Floyd. Assigning Meanings to Programs. *Mathematical Aspects of Computer Science*, 19(19-32):1, 1967.

[59] María-del-Mar Gallardo, Jesús Martínez, Pedro Merino, and Ernesto Pimentel. alpha SPIN: Extending SPIN with Abstraction. In Dragan Bosnacki and Stefan Leue, editors, *Proceedings of SPIN, Grenoble, France*, volume 2318 of *LNCS*, pages 254–258. Springer, 2002.

[60] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In Orna Grumberg, editor, *Proceedings of CAV, Haifa, Israel*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[61] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam A. Porter, and Gregg Rothermel. An Empirical Study of Regression Test Selection Techniques. In Koji Torii, Kokichi Futatsugi, and Richard A. Kemmerer, editors, *Proceedings of ICSE, Kyoto, Japan*, pages 188–197. IEEE, 1998.

[62] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. Automatically Refining Abstract Interpretations. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of TACAS, Budapest, Hungary*, volume 4963 of *LNCS*, pages 443–458. Springer, 2008.

[63] Ákos Hajdu, Tamás Tóth, András Vörös, and István Majzik. A Configurable CEGAR Framework with Interpolation-Based Refinements. In Elvira Albert and Ivan Lanese, editors, *Proceedings of FORTE, Heraklion, Greece*, volume 9688 of *LNCS*, pages 158–174. Springer, 2016.

[64] Arvind Haran, Montgomery Carter, Michael Emmi, Akash Lal, Shaz Qadeer, and Zvonimir Rakamaric. SMACK+Corral: A Modular Verifier (Competition Contribution). In Christel Baier and Cesare Tinelli, editors, *Proceedings of TACAS, London, UK*, volume 9035 of *LNCS*, pages 451–454. Springer, 2015.

[65] R. H. Hardin, R. P. Kurshan, K. L. McMillan, J. A. Reeds, and N. J. A. Sloane. Efficient Regression Verification. In R. Smedinga, M. P. Spathopoulos, and P. Kozák, editors, *Proceedings of WODES, Edinburgh, Scotland*, pages 147–150. Institute of Electronical Engineers, Computing and Control Division, 1996.

[66] Klaus Havelund and Thomas Pressburger. Model Checking JAVA Programs using JAVA PathFinder. *Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[67] Matthias Heizmann, Daniel Dietsch, Marius Greitschus, Jan Leike, Betim Musa, Claus Schätzle, and Andreas Podelski. Ultimate Automizer with Two-track Proofs (Competition Contribution). In Marsha Chechik and Jean-François Raskin, editors, *Proceedings of TACAS, Eindhoven, The Netherlands*, volume 9636 of *LNCS*, pages 950–953. Springer, 2016.

[68] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from Proofs. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of POPL, Venice, Italy*, pages 232–244. ACM, 2004.

[69] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy Abstraction. In John Launchbury and John C. Mitchell, editors, *Proceedings of POPL, Portland, OR, USA*, pages 58–70. ACM, 2002.

[70] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[71] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[72] Gerard J. Holzmann and Margaret H. Smith. A Practical Method for Verifying Event-Driven Software. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of ICSE, Los Angeles, CA, USA*, pages 597–607. ACM, 1999.

[73] Gerard J. Holzmann and Margaret H. Smith. An Automated Verification Method for Distributed Systems Software Based on Model Extraction. *IEEE Transactions on Software Engineering*, 28(4):364–377, 2002.

[74] Susan Horwitz, Thomas W. Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[75] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. TRACER: A Symbolic Execution Tool for Verification. In P. Madhusudan and Sanjit A. Seshia, editors, *Proceedings of CAV, Berkeley, CA, USA*, volume 7358 of *LNCS*, pages 758–766. Springer, 2012.

[76] Himanshu Jain, Franjo Ivancic, Aarti Gupta, Ilya Shlyakhter, and Chao Wang. Using Statically Computed Invariants Inside the Predicate Abstraction and

Refinement Loop. In Thomas Ball and Robert B. Jones, editors, *Proceedings of CAV, Seattle, WA, USA*, volume 4144 of *LNCS*, pages 137–151. Springer, 2006.

[77] Ranjit Jhala and Rupak Majumdar. Path Slicing. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of PLDI, Chicago, IL, USA*, pages 38–47. ACM, 2005.

[78] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[79] Daniel Kroening and Michael Tautschnig. CBMC — C Bounded Model Checker (Competition Contribution). In Erika Ábrahám and Klaus Havelund, editors, *Proceedings of TACAS, Grenoble, France*, volume 8413 of *LNCS*, pages 389–391. Springer, 2014.

[80] Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. SMT Techniques for Fast Predicate Abstraction. In Thomas Ball and Robert B. Jones, editors, *Proceedings of CAV, Seattle, WA, USA*, volume 4144 of *LNCS*, pages 424–437. Springer, 2006.

[81] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of CGO, San Jose, CA, USA*, pages 75–88. IEEE, 2004.

[82] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Proceedings of LPAR, Dakar, Senegal*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[83] Stefan Löwe. CPAchecker with Explicit-Value Analysis Based on CEGAR and Interpolation (Competition Contribution). In Nir Piterman and Scott A. Smolka, editors, *Proceedings of TACAS, Rome, Italy*, volume 7795 of *LNCS*, pages 610–612. Springer, 2013.

[84] Stefan Löwe. CPA-RefSel: CPAchecker with Refinement Selection (Competition Contribution). In Marsha Chechik and Jean-François Raskin, editors, *Proceedings of TACAS, Eindhoven, The Netherlands*, volume 9636 of *LNCS*, pages 916–919. Springer, 2016.

[85] Stefan Löwe, Mikhail U. Mandrykin, and Philipp Wendler. CPAchecker with Sequential Combination of Explicit-Value Analyses and Predicate Analyses (Competition Contribution). In Erika Ábrahám and Klaus Havelund, editors, *Proceedings of TACAS, Grenoble, France*, volume 8413 of *LNCS*, pages 392–394. Springer, 2014.

[86] Stefan Löwe and Philipp Wendler. CPAchecker with Adjustable Predicate Analysis (Competition Contribution). In Cormac Flanagan and Barbara König, editors, *Proceedings of TACAS, Tallinn, Estonia*, volume 7214 of *LNCS*, pages 528–530. Springer, 2012.

[87] Kenneth L. McMillan. Interpolation and SAT-Based Model Checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Proceedings of CAV, Boulder, CO, USA*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[88] Kenneth L. McMillan. Lazy Abstraction with Interpolants. In Thomas Ball and Robert B. Jones, editors, *Proceedings of CAV, Seattle, WA, USA*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

[89] Antoine Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[90] Jeremy Morse, Mikhail Ramalho, Lucas C. Cordeiro, Denis Nicole, and Bernd Fischer. ESBMC 1.22 (Competition Contribution). In Erika Ábrahám and Klaus Havelund, editors, *Proceedings of TACAS, Grenoble, France*, volume 8413 of *LNCS*, pages 405–407. Springer, 2014.

[91] Glenford J. Myers. *The Art of Software Testing (2. ed.).* Wiley, 2004.

[92] V. Natarajan and Gerard J. Holzmann. Outline for an Operational Semantics of Promela. *The SPIN Verification Systems, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. AMS*, 32:133–152, 1997.

[93] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[94] Corina S. Pasareanu, Matthew B. Dwyer, and Willem Visser. Finding Feasible Counter-examples when Model Checking Abstracted Java Programs. In Tiziana Margaria and Wang Yi, editors, *Proceedings of TACAS, Genova, Italy*, volume 2031 of *LNCS*, pages 284–298. Springer, 2001.

[95] Gregg Rothermel and Mary Jean Harrold. Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[96] Philipp Rümmer and Pavle Subotic. Exploring Interpolants. In Barbara Jobstman and Sandip Ray, editors, *Proceedings of FMCAD, Portland, OR, USA*, pages 69–76. IEEE, 2013.

[97] Jorma Sajaniemi. An Empirical Analysis of Roles of Variables in Novice-Level Procedural Programs. In *Proceedings of HCC, Arlington, VA, USA*, pages 37–39. IEEE, 2002.

[98] Roberto Sebastiani. Lazy Satisability Modulo Theories. *JSAT*, 3(3-4):141–224, 2007.

[99] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. Incremental Upgrade Checking by Means of Interpolation-based Function Summaries. In Gianpiero Cabodi and Satnam Singh, editors, *Proceedings of FMCAD, Cambridge, UK*, pages 114–121. IEEE, 2012.

[100] Pavel Shved, Mikhail U. Mandrykin, and Vadim S. Mutilin. Predicate Analysis with BLAST 2.7 (Competition Contribution). In Cormac Flanagan and Barbara König, editors, *Proceedings of TACAS, Tallinn, Estonia*, volume 7214 of *LNCS*, pages 525–527. Springer, 2012.

[101] Jiri Slaby, Jan Strejcek, and Marek Trtík. Symbiotic: Synergy of Instrumentation, Slicing, and Symbolic Execution (Competition Contribution). In Nir Piterman and Scott A. Smolka, editors, *Proceedings of TACAS, Eindhoven, The Netherlands*, volume 7795 of *LNCS*, pages 630–632. Springer, 2013.

[102] Vladimír Still, Petr Rockai, and Jiri Barnat. DIVINE: Explicit-State LTL Model Checker (Competition Contribution). In Marsha Chechik and Jean-François Raskin, editors, *Proceedings of TACAS, Eindhoven, The Netherlands*, volume 9636 of *LNCS*, pages 920–922. Springer, 2016.

[103] Gonzalez Teofilo, Jorge Diaz-Herrera, and Allen Tucker. *Computing Handbook, Third Edition: Computer Science and Software Engineering*. Chapman & Hall/CRC, 3rd edition, 2014.

[104] Alan Mathison Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Journal of Math*, 58(345-363):5, 1936.

[105] Arie van Deursen and Leon Moonen. Understanding COBOL Systems using Inferred Types. In *Proceedings of IWPC, Pittsburgh, PA, USA*, pages 74–81. IEEE, 1999.