University of Passau
Faculty of Computer Science and Mathematics

UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik

Dissertation

Towards Practical Predicate Analysis

Philipp Wendler

June 17, 2017

Supervisor:	Prof. Dr. Dirk Beyer
External Reviewer:	Prof. Dr. Bernd Fischer

Abstract

Software model checking is a successful technique for automated program verification. Several of the most widely used approaches for software model checking are based on solving first-order-logic formulas over predicates using SMT solvers, e.g., predicate abstraction, bounded model checking, *k*-induction, and lazy abstraction with interpolants. We define a configurable framework for predicate-based analyses that allows expressing each of these approaches. This unifying framework highlights the differences between the approaches, producing new insights, and facilitates research of further algorithms and their combinations, as witnessed by several research projects that have been conducted on top of this framework. In addition to this theoretical contribution, we provide a mature implementation of our framework in the software verifier CPAchecker that allows applying all of the mentioned approaches to practice. This implementation is used by other research groups, e.g., to find bugs in the Linux kernel, and has proven its competitiveness by winning gold medals in the International Competition on Software Verification.

Tools and approaches for software model checking like our predicate analysis are typically evaluated using performance benchmarking on large sets of verification tasks. We have identified several pitfalls that can silently arise during benchmarking, and we have found that the benchmarking techniques and tools that are used by many researchers do not guarantee valid results in practice, but may produce arbitrarily large measurement errors. Furthermore, certain hardware characteristics can also have nondeterministic influence on the measurements. In order to being able to properly evaluate our framework for software verification, we study the effects of these hardware characteristics, and define a list of the most important requirements that need to be ensured for reliable benchmarking. We present as solution an open-source benchmarking framework BenchExec, which in contrast to other benchmarking tools fulfills all our requirements and aims at making reliable benchmarking easy. BenchExec was already adopted by several research groups and the International Competition on Software Verification.

Using the power of BenchExec we conduct an experimental evaluation of our unifying framework for predicate analysis. We study the effect of varying the SMT solver and the way program semantics are encoded in formulas across several verification algorithms and find that these technical choices can significantly influence the results of experimental studies of verification approaches. This is valuable information for both researchers who study verification approaches as well as for users who apply them in practice. Our comprehensive study of 120 different configurations would not have been possible without our highly flexible and configurable unifying framework for predicate analysis and shows that the latter is a valuable base for conducting experiments. Furthermore, we show using a comparison against top-ranking verifiers from the International Competition on Software Verification that our implementation is highly competitive and can outperform the state of the art.

Acknowledgments

During the writing of this thesis, I have received a lot of support by many people, for which I am deeply thankful.

First of all, I want to thank my supervisor Dirk Beyer for providing me with the opportunity to write this thesis, for doing everything in his power to provide my colleagues and me with whatever we needed, for sharing his knowledge and guiding us, and for pushing us to achieve the best that we can. He had always time for me when it was needed, and many good ideas came out of our discussions.

I also want to thank Bernd Fischer, who has kindly agreed to review this thesis, and Franz Brandenburg, without whom this project would never have started.

I want to thank all the coauthors with whom I was allowed to write a paper. I enjoyed working with you, and I have learned a lot from you.

Furthermore, I want to thank my office colleagues in Passau for our joint work: Matthias Dangl, Gregor Endler, Karlheinz Friedberger, Peter Häring, Malte Rosenthal, Andreas Stahlbauer, and especially Stefan Löwe, with whom it was a pleasure to share an office for several years. A big thank you goes to our assistants Eva Veitweber and Eva Reichhart, who kept the whole chair running smoothly by taking care of all the administrative duties for us. I always had a good time together with all of you, and also with the colleagues from the neighboring chairs.

The last months' work of this thesis was done at LMU Munich, where I want to thank Marianne Diem, Anton Fasching, Rolf Hennicker, Annabelle Klarl, and Philip Mayer for giving me a warm welcome and a lot of initial support.

I want to thank our student assistants Alexander Driemeyer, Thomas Lemberger, Sebastian Ott, and Thomas Stieglmaier, and all those who I supervised for their Bachelor's or Master's thesis. You contributed a lot of high-quality work to our projects. The same holds for all other developers and users of CPAchecker and BenchExec.

Last but not least, I want to thank my fiancée Teresa, my family, and my friends, who have supported me and accepted that I could often spend less time with them than desired.

Contents

1. Introduction 1

I. A Flexible Domain Based on Predicates 9

2. Motivation 13

3. Background 19

4. Predicate CPA 31

5. Applications 57

6. Implementation in CPAchecker 73

7. Related Work 81

II. Reliable Benchmarking 89

8. Motivation 93

9. Requirements for Reliable Benchmarking 99

10. Limitations of Existing Methods 105

11. Impact of Hardware Characteristics on Parallel Tool Executions 111

12. State-of-the-Art Benchmarking with Cgroups and Containers 123

13. BenchExec: A Framework for Reliable Benchmarking 131

14. Related Work 143

III. Experimental Evaluation **149**

15. Experiment Setup **153**

16. Comparison of SMT Solvers and Theories **159**

17. Comparison with State of the Art **175**

18. Summary and Future Work **181**

Bibliography **185**

Appendix **205**

A. Machine Architectures with Hyperthreading and Nonuniform Memory Access **207**

B. Listings **211**

List of Figures

1.1. Contributions of this thesis 3

4.1. Components of the Predicate CPA 32
4.2. Refinement steps 37
4.3. Valid combinations of abstract facts and refinement strategies . . . 43
4.4. Statement `*p = *p + *q` encoded in SMTLIB with a byte-wise memory array 54

5.1. Classification of verification approaches 66
5.2. Sequential combination of algorithms as used in SV-COMP'15 submission 72

6.1. Architecture of CPAchecker with components for predicate analysis 74

9.1. Requirements for reliable benchmarking 100

11.1. Scatter plot showing the influence of hyperthreading 114
11.2. Plot showing the influence of Turbo Boost, shared Level 3 cache, and shared memory bandwidth 116
11.3. Plot showing the influence of using multiple CPUs 117
11.4. Scatter plot showing the influence of multiple CPUs 118

13.1. Resource control and process isolation by `runexec` 133
13.2. Benchmarking with `benchexec` 135

16.1. Quantile plots for CPU time of bitprecise SMT theories across all available SMT solvers and algorithms 161
16.2. Quantile plots for CPU time of SMT theories with linear arithmetic across all available SMT solvers and algorithms 166
16.3. Quantile plots for CPU time of SMT theories with linear arithmetic using unsound overflows across all available SMT solvers and algorithms 170

16.4. Quantile plots for CPU time of SMT theory combinations QF_UFBVFP and QF_UFLIRA 173

17.1. Quantile plots for CPU time of verifiers from SV-COMP'17 and configurations of our framework for predicate-based software verification 177

A.1. Example for a machine with a NUMA architecture: 2 AMD Opteron 6380 CPUs, each with two groups of four modules of two cores and 69 GB (64 GiB) of RAM 208

A.2. Example for a machine with a NUMA architecture: 2 Intel Xeon E5-2650 v2 CPUs, each with eight physical cores with hyperthreading and 68 GB (63 GiB) of RAM 209

List of Tables

4.1. Classification of abstract facts according to fulfilled properties . . . 40
4.2. Possible combinations of SMT theories for encoding program semantics . . . 51

5.1. Configurations of Predicate CPA for common SMT-based approaches 60

6.1. Interpolating SMT solvers . . . 76

16.1. Results for bitprecise SMT theories across all available SMT solvers and algorithms . . . 160
16.2. Results for SMT theories with linear arithmetic across all available SMT solvers and algorithms . . . 164
16.3. Results for SMT theories with linear arithmetic (encoding with unsound overflows) across all available SMT solvers and algorithms 168
16.4. Comparison of SMT theory combinations QF_UFBVFP and QF_UFLIRA . . . 172

17.1. Results for verifiers from SV-COMP'17 and configurations of our framework for predicate-based software verification . . . 176

List of Algorithms

3.1. CPA+$(\mathbb{D}, e_{INIT}, \pi_{INIT})$ 22

5.1. CPA++$(\mathbb{D}, \mathsf{reached}, \mathsf{waitlist}, \mathsf{abort})$ 58
5.2. Iterative-Deepening *k*-Induction with Invariants 62
5.3. CEGAR$(\mathbb{D}, e_{INIT}, \pi_{INIT})$ for CPAs 64

List of Listings

B.1. Example for using module `runexec` from a Python program . . . 211
B.2. Example for an XML file as input for program `benchexec` 212

1. Introduction

Software has become an extremely important part of our lives, controlling many safety-critical systems, such as power plants, planes, trains, and cars. In these applications it is paramount to avoid bugs, but also in less critical systems such as the software running on our personal machines or on servers, bugs can have severe consequences. Testing [197] is a widely used approach for detecting bugs, but is not able to find all bugs or prove absence of bugs. Manual [117, 141] or semiautomatic [181, 201] verification is sometimes used in the development of safety-critical systems, but is expensive because of the high effort and the need for specialized experts, making it unsuitable for many software projects. Automatic software verification is an attempt to fill the gap and provide higher confidence than testing with less effort than manual verification: Given an input program and a specification, a verification tool should determine automatically whether the specification holds for the program, or whether there exists a counterexample. Of course, because of the undecidability of the problem in general, the result may also be inconclusive, for example because the verification tool gives up or exceeds the available time and memory resources. Nevertheless, approaches such as software model checking [83, 85, 204] have proven useful for example in the verification of kernel drivers [11, 13, 50, 166] or other safety-critical software [56], however, they are not yet used widely by software developers. Our goal is to develop approaches and tools for automatic software verification in order to bring it further towards a practically useful technique that allows developers to build better software.

Progress in the field of automatic software verification depends among others on a good theoretical understanding of approaches and the possibility to effectively evaluate them in practice, i.e., by performing sound experiments with regard to their effectivity and performance. We have identified three kinds of problems in that regard.

First, we note that the presentation of verification approaches often differs substantially, even if the approaches are related. This is not a problem per se, but such superficial differences can obscure the actual conceptual differences, and make it hard to understand what the core ideas of a given approach are and to learn from the differences and relations of approaches.

Second, practical experimentation for evaluating approaches is often difficult. Some approaches are implemented only in tools that are not freely available or in academic research prototypes, which may, for example, support only a subset of the input language and are not applicable to commonly used benchmark sets. Such tools might also miss technical optimizations, and this can lead to flawed experiments and wrong scientific conclusions that cannot be confirmed when a better implementation is used [206]. Furthermore, one often needs to compare several approaches against each other, but this can be problematic even if a tool with a mature implementation exists for each of them. Different tools typically differ in a lot of factors that can influence the performance heavily but are not relevant to a comparison of their core approaches, for example in their runtime environments, which libraries they use (e.g., as solver), and which optimizations they have implemented. Thus, a comparison of different approaches implemented in different tools may also lead to wrong conclusions.

Third, for valid performance experiments that allow us to draw meaningful conclusions we also need to ensure that the benchmarking is reliable and its results are precise and accurate. We have identified several problems in the tools and techniques used for benchmarking in our field that can lead to unnoticed measurement errors of arbitrary size. One instance of such a problem almost invalidated the results of the 2nd International Competition on Software Verification (SV-COMP'13), highlighting that the validity of results depends as much on the benchmarked tool as well as on the benchmarking tool, but the latter is often given too little attention. Furthermore, the complex hardware characteristics of today's machines need to be taken into account when benchmarking to ensure replicable results. Because the evaluation of new contributions in the field of automatic software verification is to a large degree based on performance experiments, threats to the validity of these experiments must be taken seriously.

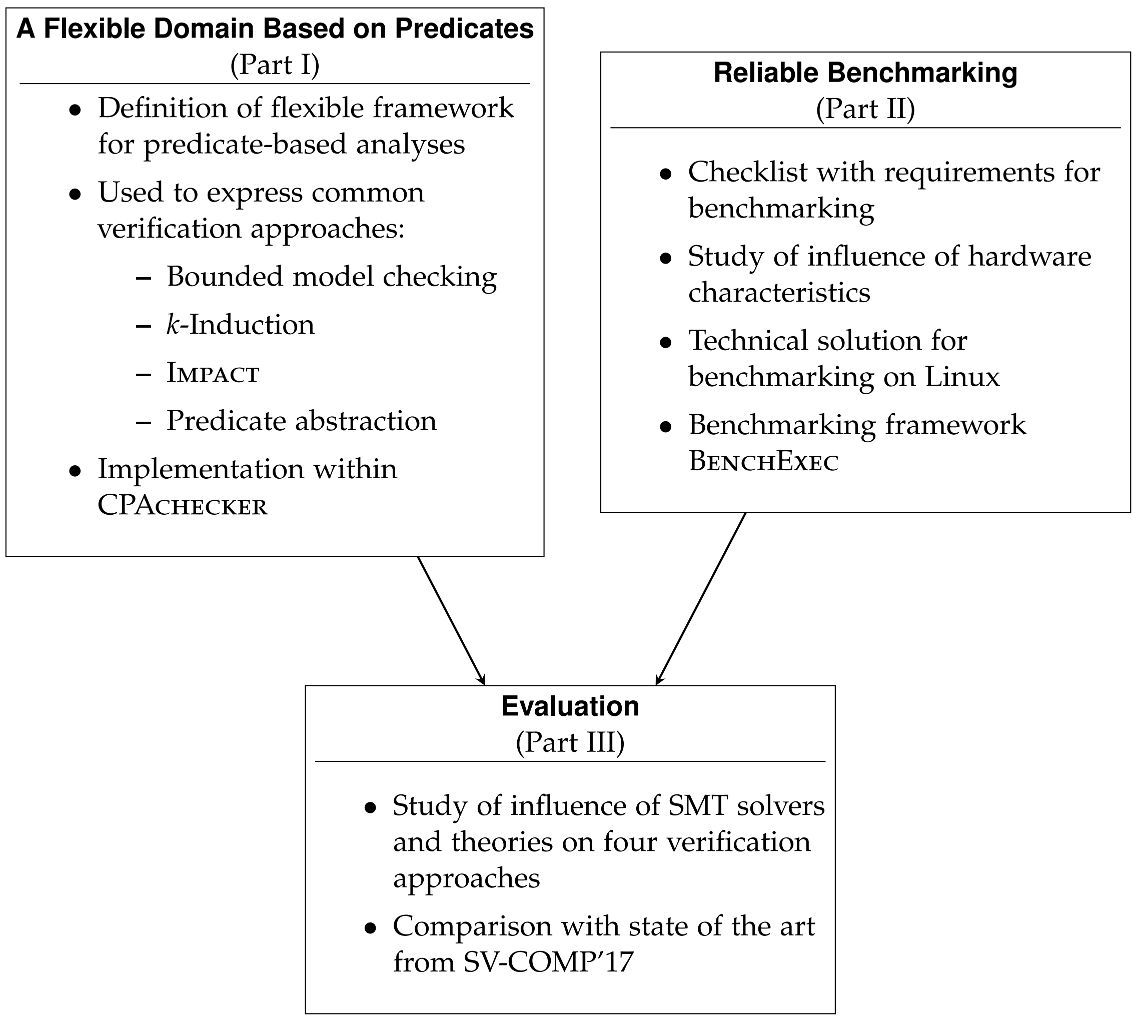

A Flexible Domain Based on Predicates
(Part I)
Definition of flexible framework for predicate-based analyses
Used to express common verification approaches:
Bounded model checking
k-Induction
IMPACT
Predicate abstraction
Implementation within CPACHECKER
Reliable Benchmarking
(Part II)
Checklist with requirements for benchmarking
Study of influence of hardware characteristics
Technical solution for benchmarking on Linux
Benchmarking framework BENCHEXEC
Evaluation
(Part III)
Study of influence of SMT solvers and theories on four verification approaches
Comparison with state of the art from SV-COMP'17

Figure 1.1.: Contributions of this thesis

1.1. Contributions

The goal of this thesis is to provide a solution to the aforementioned problems. We focus on software model checking based on predicates over program variables, which is an important class of approaches for software verification. This allows us to leverage the power of modern solvers for satisfiability modulo theory (SMT) to build highly efficient and powerful analyses. Well-known examples for predicate-based analyses are predicate abstraction [125] and bounded model checking (BMC) [54], which are implemented in many successful tools [13,35,41,86,87,96,116].

The main contribution of this thesis is the definition of a flexible and unifying framework for approaches for automatic software verification that are based on predicates. In addition to that, we also provide an explanation how to express four common verification approaches using our framework, a mature implementation of this framework as a highly configurable predicate analysis inside the tool CPAchecker[1], a technical solution for reliable performance benchmarking with our benchmarking framework BenchExec[2], and an experimental study using our flexible framework that provides new insights. Figure 1.1 also lists these contributions and their structure in this thesis.

Together, these contributions make it easy to design, implement, experiment with, and use predicate-based analyses for software verification, hopefully allowing this field to thrive even more in the future and to pave the way to making software verification a part of every developer's toolbox.

In particular, a unifying theoretical framework for analyses often helps to gain new insights when approaches are expressed within the framework, and thus their key differences with respect to other existing approaches are highlighted. This might allow for example understanding why some approach performs better than another, and might ease the design of and experiments with new combinations of approaches. Using an existing well-known framework might also make it easier to express some new approach and to read and understand it. We define such a framework for predicate-based analyses based on our previous work of a domain for predicate abstraction adjustable-block encoding [42], and we use it to express four of the most common approaches for software verification in it: BMC[3] [54], *k*-induction[4] [217], predicate abstraction[5] [125] with lazy abstraction [140], and Impact-style lazy abstraction with interpolations[6] [190]. Our paper describing an earlier version of this framework was published at FMCAD'12 [52] and an updated version is to appear in the Springer journal JAR [33].

A mature implementation of an approach is not only useful when attempting to apply the approach in practice, but also for researchers [6] because it is a valuable base for experimental evaluation of further ideas and methods that

[1] `https://cpachecker.sosy-lab.org`
[2] `https://github.com/sosy-lab/benchexec`
[3] Implemented for example in Cbmc [86], Esbmc [96], and Llbmc [116].
[4] Implemented for example in 2ls [64], DepthK [207], Esbmc [195], and PKind [159].
[5] Implemented for example in Blast [35], F-Soft [148], Satabs [87], and Slam [13].
[6] Implemented for example in Ufo [4] and Wolverine [175].

build on this approach. A configurable and flexible framework also allows implementing new approaches on top of it more easily. We have implemented our approach in the software verification framework CPAchecker [41], which allows verification of sequential C programs. This implementation was submitted to the 1st International Competition on Software Verification (SV-COMP'12) and won a silver medal in the category *Overall* as well as a gold medal in the category *ControlFlowInteger*. Our paper describing the competition contribution was published at TACAS'12 [187].

For being able to properly evaluate our framework for software verification, we need to reliably measure performance in a replicable way. We begin with defining a checklist of requirements for reliable benchmarking and studying the potential influences of hardware characteristics. This allows researchers to properly setup benchmarking experiments and reviewers to assess the quality of benchmarking results. Then we explain a solution for reliable benchmarking on Linux systems, which uses benchmarking containers based on cgroups and namespaces. Furthermore, we have implemented an open-source benchmarking framework BenchExec, which, in contrast to other benchmarking tools, implements this solution and fulfills all our requirements for reliable benchmarking. BenchExec is not only suited for the evaluation performed for this work, but was already adopted by several research groups and the International Competition on Software Verification (SV-COMP). A description of our benchmarking requirements and BenchExec was published in the proceedings of the conference SPIN'15 [46] and an extended version was accepted with minor revisions for publication in the Springer journal STTT [49].

Using BenchExec as benchmarking framework we experimentally evaluate our framework for software verification. We show its usefulness for performing meaningful experiments by conducting a comprehensive study about the influence of SMT solvers and theories on experimental results for various verification approaches. The flexibility of our framework allows us to compare 120 different configurations, which took 3 620 days of CPU time across 671 280 verification runs. This study shows that the choice of the SMT solver can have a significant influence and create a bias against specific verification approaches. This is a valuable insight that means that such technical choices need to be carefully considered by researchers when performing experiments with predicate-based verification approaches. Furthermore, we demonstrate the effectivity and efficiency of the implementation of our framework for software verification by comparing it against top-ranking software verifiers from the 6th International Competition on Software Verification (SV-COMP'17), showing that it is competitive with the state of the art in software verification.

1.1.1. Availability of Data and Tools

All code that belongs to this thesis is freely available in the open-source tools CPAchecker and BenchExec, which are both licensed under the permissive Apache 2.0 license. The raw data for all experimental results as well as everything that is necessary to repeat the experiments is also available online.[7]

1.1.2. Achievements

The contributions of this thesis have already shown their value in a number of successful research projects besides what is described in this work. The predicate domain and its implementation were used in projects on conditional model checking [36], witness validation [29, 30], block-abstraction memoization [233], combining *k*-induction with auxiliary invariants to make it usable for software verification [31], symbolic execution with CEGAR [43], verification of recursive programs [75], regression verification [45], refinement selection [47], local policy iteration [164], and a study of algorithms for software model checking [28]. Its implementation has proven its effectiveness and efficiency in six years of participation in SV-COMP, where it has (alone or in combination with other analyses) helped to win 12 gold medals, 16 silver medals, and 16 bronze medals.[8] These achievements of CPAchecker in SV-COMP have also been awarded by the Kurt Gödel Society with a Gödel medal at the Vienna Summer of Logic 2014.[9] Furthermore, CPAchecker with its predicate analysis is routinely used in the

[7] `https://www.sosy-lab.org/research/phd/wendler`

[8] Counting the following submissions, all of which make use of (parts of) the work presented here:
CPAchecker-ABE [187] and CPAchecker-Memo [232] for SV-COMP'12 [19],
CPAchecker-Explicit [184] and CPAchecker-SeqCom [231] for SV-COMP'13 [20],
CPAchecker [186] for SV-COMP'14 [21],
CPAchecker [100], CPArec [76], and Unbounded Lazy-CSeq [199] for SV-COMP'15 [22], CPA-BAM [119], CPA-kInd, CPA-RefSel [185], CPA-Seq, LPI [162], and UL-CSeq for SV-COMP'16 [23], and CPA-BAM-BnB [7], CPA-kInd and CPA-Seq for SV-COMP'17 [24].

[9] `https://cpachecker.sosy-lab.org/achieve.php#awards`

Linux Driver Verification project[10] [166] to analyze device drivers of the Linux kernel [165], and has helped to identify and fix a range of bugs in the Linux kernel[11]. Predicate-based analyses building on the concepts and the implementation described here are part of the default configuration of CPAchecker for validating witnesses [29, 30], and are used as one of two witness validators to validate results in SV-COMP [23, 24].

The benchmarking framework BenchExec is the technical foundation of SV-COMP [23], one of the largest competitions in the field of computer-aided verification with 32 participants and more than 400 000 benchmark runs in its latest instance [24].

1.2. Structure

This thesis is structured as follows. In Part I we present a flexible domain for software verification that is based on predicates. We provide a formal definition of the framework in Chapter 4 and explain how to express various algorithms for software model checking within it in Chapter 5. Chapter 6 describes our implementation in the open-source software-verification framework CPAchecker.

Part II describes how to perform reliable benchmarks for experimental evaluations. We present six important requirements for benchmarking in Chapter 9, and in Chapters 10 and 11 we motivate these requirements by showing what kinds of measurement errors can occur otherwise. Chapter 12 provides a solution for how to implement a reliable benchmarking tool on Linux, and in Chapter 13 we present our open-source tool BenchExec, which implements this solution and fulfills all the stated requirements.

Part III builds upon Part I and Part II by using our benchmarking solution to perform an experimental study of our unifying framework for predicate analysis. In Chapter 16 we study the effect of varying the SMT solver and the SMT theories that are used for encoding program semantics across the verification approaches that we have expressed in our unifying framework. Chapter 17 provides a comparison of our implementation with top-ranking software verifiers from SV-COMP'17.

[10] `http://linuxtesting.org/ldv`
[11] `https://cpachecker.sosy-lab.org/achieve.php#bugsfound`

Part I.

A Flexible Domain Based on Predicates

Contents

2. Motivation **13**
2.1. Overview 16
2.2. Restrictions 16

3. Background **19**
3.1. Program Representation 19
3.2. Configurable Program Analysis 20
3.2.1. CPA Algorithm 21
3.2.2. Composite CPA 22
3.2.3. Basic CPAs 23
3.3. Predicate Abstraction 23
3.3.1. Counterexample-Guided Abstraction Refinement 24
3.3.2. Lazy Abstraction 24
3.3.3. Adjustable-Block Encoding 25
3.4. IMPACT (Lazy Abstraction with Interpolants) 26
3.4.1. IMPACT with Forced Covering 27
3.5. Bounded Model Checking and *k*-Induction 28
3.6. CPAchecker 29

4. Predicate CPA **31**
4.1. Abstract Domain, Precisions, and CPA Operators 31
4.1.1. Abstract States 31
4.1.2. Precisions 33
4.1.3. Transfer Relation 33
4.1.4. Merge Operator 34
4.1.5. Stop Operator 34
4.1.6. Precision-Adjustment Operator 35

4.2. Refinement 37
4.2.1. Abstract-Counterexample Construction 37
4.2.2. Feasibility Check 38
4.2.3. Abstract Facts and Discovery Strategies 38
4.2.4. Refinement Strategies 43
4.3. Forced Covering 47
4.4. Encoding C Semantics 48
4.4.1. Nonlinear Arithmetic 51
4.4.2. Pointer Accesses 53
4.4.3. Heap Memory 54

5. Applications 57
5.1. An Extended CPA Algorithm 57
5.2. Unifying SMT-Based Approaches for Software Verification 59
5.2.1. Bounded Model Checking 59
5.2.2. *k*-Induction 61
5.2.3. Lazy Predicate Abstraction 64
5.2.4. Lazy Abstraction with Interpolants (IMPACT) 65
5.3. Comparison of SMT-based Approaches for Software Verification . 66
5.4. Configurability and Extensions 68
5.4.1. Adjustable-Block Encoding for IMPACT 69
5.4.2. Flexible Bounded Analyses and Counterexample Checks . 69
5.4.3. Further Configuration Options 70
5.4.4. Sequential Combinations of Configurations 70

6. Implementation in CPACHECKER 73
6.1. Discovery Strategies for Abstract Facts 73
6.2. Strongest-Postcondition Operator 75
6.3. SMT Solvers 75
6.3.1. Comparison 77
6.3.2. Integration in CPACHECKER 78

7. Related Work 81
7.1. SMT-Based Algorithms for Software Model Checking 81
7.2. Extensions of the Studied Approaches 84
7.3. Software Verifiers 86

2. Motivation

Many approaches for software model checking based on predicates and SMT solving have been developed and studied in the last 20 years, for example predicate abstraction [125, 140], bounded model checking [54] with and without *k*-induction [217], lazy abstraction with interpolants [190], slicing abstractions [68, 114], and property-directed reachability [55, 62, 78]. Most of these approaches have been implemented in software-verification tools [1], and some of them are also used in practice for verifying software, e.g., for verification of kernel drivers [11, 166]. However, the current state can be described as fragmented: multiple approaches and tools coexist without much connection. A unifying framework for all kinds of predicate-based analyses, which is highly configurable and flexible, and for which a mature implementation exists, could be an important step for further research in this area by bringing the following advantages.

A unifying theoretical framework for analyses often helps to gain new insights when approaches are expressed within the framework, and thus their key differences with respect to other existing approaches are highlighted. This might allow for example understanding why some approach performs better than another one, and designing and experimenting with new combinations of approaches. Using an existing well-known framework might also make it easier to express and implement some new approach, and to read and understand it.

An example where a theoretical framework for software verification has proven useful for research is configurable program analysis (CPA) [39]. This framework has unified software model checking and (lattice-based) program analysis, and different abstract domains have been expressed within it, including domains for predicate abstraction [42], explicit-value analysis [44], BDDs [51], and intervals [32]. The CPA concept has facilitated research with regard to the combination

[1] e.g., Blast [35], Cbmc [86], CPAchecker [41], Esbmc [96], Kratos [79], Llbmc [116], Satabs [87], Slam [13], Ufo [4], Ultimate Kojak [115], and Wolverine [175]

of abstract domains [8, 31, 44, 47] and the development of new verification approaches, e.g., local policy iteration [164].

A common framework also increases the validity of experimental evaluations that compare the approaches that are expressed within it, because it minimizes all other factors that can influence the experimental results. Such factors include not only purely technical implementation differences, such as their runtime environment or the libraries they use, e.g., as solvers, but also conceptual improvements, for example whether techniques such as the encoding of blocks of several program operations at once [10, 27, 42] or loop acceleration [14, 57, 61, 151, 173] are integrated.

Furthermore, being able to easily switch between and combine several verification approaches can be important in practice. Even the best existing verification approaches vary in their effectiveness if applied to verification tasks with different characteristics [28], such as which kind of property is to be verified, whether the analyzed program is more control-flow oriented or data heavy, whether precise overflow semantics, recursion, concurrency, or unbounded heap structures need to be supported, etc. This is shown for example by the results of SV-COMP'16 [23]: Out of the 8 regular (non-meta) categories, 7 were won by different participants, and the only submission that won 2 categories (CPA-Seq) is actually a configuration that combines 8 individual approaches. This means that users of verifiers might want to experiment with different approaches in order to find an approach (or even a combination of approaches) that is well-suited for their uses cases, something that is made more difficult or even impractical if every approach is implemented in a different tool.

Of course, a mature and efficient implementation of an approach is important for users that want to apply this approach in practice. However, the same is true also for researchers that want to study a given approach using performance benchmarking [6], a common method for evaluation in the field of software verification. Such experimental evaluations may be based on complex source code from real-world software, which a purely prototypical academic verifier might not be able to parse.[2] The quality of an implementation with regard to efficiency is also relevant for a valid experimental evaluation. Experience shows that missing optimizations can influence performance results in a way

[2] For example, the Linux kernel, which is often used as case study and source of verification tasks [19, 121, 165, 166, 196], uses many GCC-specific language extensions.

that researchers draw conclusions that can no longer be upheld once these purely technical optimizations get implemented [206]. And last but not least, the challenges that occur while developing mature tools and applying them in practice can lead to new insights that foster further scientific progress [10, 234].

A high flexibility and configurability of an approach is valuable because it makes research on new variants easier. For example, in a framework that unifies several approaches and is also highly configurable, one can experiment with hybrid approaches that combine key elements from multiple of the existing approaches.

Furthermore, a theoretically well-suited configuration could be unusable in practice. For example, it could happen that in some situations the selection of possible SMT solvers is restricted, maybe because of license or platform incompatibilities, and the available SMT solvers do not support all necessary features. Trade-offs between performance and correctness also may need to be made. For example, handling overflows and bit-manipulation operators needs support for the theory of bitvectors in the SMT solver, and this theory and especially interpolation for it are not as widely available and stable as the theories of linear arithmetic over integers and reals. In some cases, e.g., if the property to check is conformance to a specific function-calling protocol, a user might consider the risk of overflow-related specification violations low enough to use a faster but unsound approximation with linear arithmetic instead of a bitprecise verification. Thus, an analysis that aims to be useful for a broad spectrum of users should be flexible enough to support such variants.

Our goal is to define a configurable and flexible framework for predicate-based approaches that is helpful both in theory (by unifying approaches and thus simplifying their development and study) as well as in practice (by being customizable for different use cases). In addition, a mature and efficient implementation of this framework should allow reliable scientific experiments and application in practice of the approaches that are integrated now and in the future.

2.1. Overview

In the remainder of this part, we present the following contributions:

- We formally define the core of our framework, a flexible abstract domain based on predicates expressed as a configurable program analysis [39] (Chapter 4). This CPA makes use of SMT solving and interpolation [97], has support for lazy abstraction [140] and adjustable-block encoding (ABE) [42], and can be used for counterexample-guided abstraction refinement (CEGAR) [84].
- We show how to express four common approaches for software model checking using our abstract domain: predicate abstraction [125], lazy abstraction with interpolants (IMPACT) [190], bounded model checking [54], and *k*-induction [217] (Chapter 5). This allows us to define new variants and combination of these approaches.
- We describe our implementation of this framework for predicate-based analyses in the open-source software verification framework CPACHECKER [41], which supports the verification of C programs (Chapter 6).

Later on, in Part III, we will provide an experimental evaluation of this framework using the benchmarking techniques from Part II.

Our abstract domain is based on an existing CPA for predicate abstraction with ABE [42]. A preliminary version of the Predicate CPA and the comparison of predicate abstraction with IMPACT have been published previously [52], and a publication that contains most of this part of this thesis is to appear soon [33]. Our implementation is also explained in the description of our first competition contribution to SV-COMP [187].

2.2. Restrictions

Our goal is only the verification of safety properties [176] of sequential programs. However, there exist techniques that may allow to adapt our approach to the verification of liveness properties [215].

In this thesis, we discuss only the verification of nonrecursive C programs. Verification of recursive programs can be reduced to an iterative verification of a series of nonrecursive programs [75]. Furthermore, specifically for CPA-based analyses there exist the possibility to use block-abstraction memoization [233] to add support for recursion [118]. The core of our framework including all algorithms is not specific to C, and can be extended to other imperative and related programming languages by replacing only the operator that is responsible for encoding the semantics of program operations into formulas. The same holds true for our implementation, if a parser frontend for the respective language is added to CPAchecker.

3. Background

Before we define our framework, we provide basic definitions from the literature [35] and describe the existing approaches for predicate-based software model checking that we will unify later (descriptions taken from [52]).

3.1. Program Representation

For simplicity, we initially restrict our presentation to programs in a simple imperative programming language without functions, in which all basic operations are assume operations or assignments, and all variables are integers.[1] Such a program can be represented using a *control-flow automaton* (CFA), which is a directed graph with program operations attached to its edges. A CFA $A = (L, l_{INIT}, G)$ consists of a set L of *program locations*, an initial location $l_{INIT} \in L$ that represents the program entry point, and a set $G \subseteq (L \times Ops \times L)$ of edges between program locations, each labeled with an operation that is executed when the control flows along the edge. The set of all program variables that occur in the operations of a CFA is denoted by X. A *concrete data state* $c : X \to \mathbb{Z}$ is a mapping from program variables to integers. A set of concrete data states is called *region*. We represent regions using first-order formulas ψ over variables from X such that the set $[\![\psi]\!]$ of concrete data states that is represented by ψ is defined as $\{c \mid c \models \psi\}$. A *concrete state* $(c, l) : (X \to \mathbb{Z}) \times L$ is a pair of a concrete data state and a location.

An operation $op \in Ops$ can either be an assignment of the form $x := e$ with a variable $x \in X$ and an arithmetic expression e over variables from X, or an assume operation $[p]$ with a predicate p over variables from X. The semantics of an operation op is defined by the *strongest-postcondition operator* $\mathsf{SP}_{op}(\cdot)$. For a formula ψ and an assignment $x := e$, it is defined as $\mathsf{SP}_{x:=e}(\psi) = \exists \hat{x} : \psi_{[x \to \hat{x}]} \wedge (x = e)$, and for an assume operation $[p]$ as $\mathsf{SP}_{[p]}(\psi) = \psi \wedge p$.

[1] Later on in Sect. 4.4 we will discuss how to extend this to the C programming language.

Note that in the implementation we can avoid the existential quantifier in the strongest-postcondition operator for assignments by skolemization: Similarly to a static single-assignment (SSA) form [99] like it is used by compilers, for each program variable an *SSA index* counter is added that is incremented on every assignment to the variable, and all variable accesses use the variable qualified with the current index value. If this encoding is used, program variables in formulas returned by $\mathsf{SP}(\cdot)$ are always qualified by their SSA index, whereas program variables in formulas representing regions are still used in the unqualified variant. Thus, an appropriate renaming (adding or removing SSA indices) needs to be done whenever these kinds of formulas are combined, e.g., in a conjunction.

A *path* $\sigma = \langle (l_i, op_i, l_j), (l_j, op_j, l_k), \ldots, (l_m, op_m, l_n) \rangle$ is a sequence of consecutive edges from G. A path is called *program path* if it starts in the initial location l_{INIT}. The semantics of a path is defined by the iterative application of $\mathsf{SP}_{op}(\cdot)$ for each operation of the path: $\mathsf{SP}_{\sigma}(\psi) = \mathsf{SP}_{op_m}(\ldots(\mathsf{SP}_{op_i}(\psi))\ldots)$. A path σ is called *feasible* if $\mathsf{SP}_{\sigma}(true)$ is satisfiable, and infeasible otherwise. A location l is called *reachable* if there exists a feasible path from l_{INIT} to l.

A *verification task* (for safety verification) consists of a CFA $A = (L, l_{INIT}, G)$ and an error location $l_{ERR} \in L$[2], with the goal to show that l_{ERR} is unreachable in A, or to find a feasible error path (i.e., a feasible program path to l_{ERR}) otherwise.

3.2. Configurable Program Analysis

A configurable program analysis (CPA) [39] specifies the abstract domain that is used for a program analysis. By using the concept of CPAs we can define the abstract domain independently from the analysis algorithm: the CPA algorithm is an algorithm for reachability analysis that can be used with any CPA. Furthermore, CPAs can be combined to compositions of CPAs. The CPAs defined in this work make use of the extension CPA+ (dynamic precision adjustment) [40], but for simplicity we continue to name them CPAs.

A CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ consists of an abstract domain D, a set Π of precisions, a transfer relation $\rightsquigarrow$, and the operators merge, stop, and prec. The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of a set C of concrete states,

[2] We use a single error location for ease of presentation, our implementation allows more complex specifications using monitor automata [26].

a semilattice $\mathcal{E} = (E, \sqsubseteq)$ over a set E of abstract-domain elements (i.e., abstract states) and a partial order $\sqsubseteq$ (the join $\sqcup$ of two elements and the join $\top$ of all elements are unique), and a concretization function $[\![\cdot]\!]$ that maps each abstract-domain element to the represented set of concrete states. We call an abstract state $e \in E$ an *abstract error state* if it represents a concrete state at the error location l_{ERR}, i.e., if $\exists c \in (X \to \mathbb{Z}) : (c, l_{ERR}) \in [\![e]\!]$. The transfer relation $\rightsquigarrow \subseteq E \times E \times \Pi$ computes abstract successor states under a precision. The merge operator $\mathsf{merge} : E \times E \times \Pi \to E$ specifies if and how to merge two abstract states when control flow meets under a given precision. The stop operator $\mathsf{stop} : E \times 2^E \times \Pi \to \mathbb{B}$ determines whether an abstract state is covered by a given set of abstract states. The precision-adjustment operator $\mathsf{prec} : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$ allows adjusting the analysis precision dynamically depending on the current set of reachable abstract states. The operators merge, stop, and prec can be chosen appropriately to influence the abstraction level of the analysis. Common choices include $\mathsf{merge}^{sep}(e, e', \pi) = e'$ (which does not merge abstract states), $\mathsf{stop}^{sep}(e, R, \pi) = (\exists e' \in R : e \sqsubseteq e')$ (which determines coverage by checking whether the given abstract state is less than or equal to any other reachable abstract state according to the semilattice), and $\mathsf{prec}^{id}(e, \pi, R) = (e, \pi)$ (which keeps abstract state and precision unchanged).

3.2.1. CPA Algorithm

CPAs can be used by the CPA algorithm for reachability analysis (cf. Algorithm 3.1), which gets as input a CPA and an initial abstract state with precision. The algorithm does a classic fixed-point iteration by looping until the set waitlist is empty (all abstract states have been completely processed) and returns the set of reachable abstract states. In each iteration, the algorithm takes one abstract state e with precision π from the waitlist, passes them to the precision-adjustment operator prec, computes all abstract successors, and processes each of the successors. The algorithm checks if there is an existing abstract state with precision in the set of reached states with which the successor abstract state is to be merged (e.g., at join points where control flow meets after completed branching). If this is the case, then the new, merged abstract state with precision substitutes the existing abstract state with precision in both sets reached and waitlist. The stop operator en-

Algorithm 3.1 $\mathsf{CPA+}(\mathbb{D}, e_{INIT}, \pi_{INIT})$, taken from [40]

```
Input: a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$,
    where $E$ denotes the set of elements of the semilattice of $D$,
    and an initial abstract state $e_{INIT} \in E$ with precision $\pi_{INIT} \in \Pi$,
Output: a set of reachable abstract states
Variables: two sets reached and waitlist of elements of $E \times \Pi$
 1: $\mathsf{reached} := \{(e_{INIT}, \pi_{INIT})\}$
 2: $\mathsf{waitlist} := \{(e_{INIT}, \pi_{INIT})\}$
 3: while $\mathsf{waitlist} \neq \emptyset$ do
 4:   pop $(e, \pi)$ from waitlist
 5:   $(\hat{e}, \hat{\pi}) := \mathsf{prec}(e, \pi, \mathsf{reached})$                // Adjust the precision.
 6:   for all $e'$ with $\hat{e} \rightsquigarrow (e', \hat{\pi})$ do
 7:     for all $(e'', \pi'') \in \mathsf{reached}$ do
 8:       $e_{new} := \mathsf{merge}(e', e'', \hat{\pi})$     // Combine with existing abstract state.
 9:       if $e_{new} \neq e''$ then
10:         $\mathsf{waitlist} := \big(\mathsf{waitlist} \cup \{(e_{new}, \hat{\pi})\}\big) \setminus \{(e'', \pi'')\}$;
11:         $\mathsf{reached} := \big(\mathsf{reached} \cup \{(e_{new}, \hat{\pi})\}\big) \setminus \{(e'', \pi'')\}$;
12:     if not $\mathsf{stop}(e', \{e \mid (e, \cdot) \in \mathsf{reached}\}, \hat{\pi})$ then   // Add new abstract state?
13:       $\mathsf{waitlist} := \mathsf{waitlist} \cup \{(e', \hat{\pi})\}$
14:       $\mathsf{reached} := \mathsf{reached} \cup \{(e', \hat{\pi})\}$
15: return $\{e \mid (e, \cdot) \in \mathsf{reached}\}$
```

sures that a new abstract state is inserted into the work sets only if this is needed, i.e., the abstract state is not already covered by an abstract state in the set reached.

3.2.2. Composite CPA

Several CPAs can be combined (Composite pattern) using a *Composite CPA* [39]. The abstract states of the Composite CPA are tuples of one abstract state from each component CPA, the precisions of the Composite CPA are tuples of one precision from each component CPA, and the operators of the Composite CPA delegate to the component CPAs' operators accordingly.

The effect of such a combination of CPAs is that all used CPAs work together in eliminating infeasible paths during the program analysis: one CPA might be able to prove some specific paths infeasible, whereas other CPAs might rule out other infeasible paths. The analysis will only find paths which all used CPAs agree to be feasible. Note that this effect already occurs without any form of communication or information exchange between the component CPAs, and neither does any of the component CPAs need to know anything about the others. However, for an even higher precision, information exchange is possible if desired using the strengthen operator $\downarrow$ [39] and precision-adjustment operator prec [40] of the Composite CPA.

3.2.3. Basic CPAs

The possibility to combine CPAs by using a Composite CPA allows us to separate different concerns: we extract certain common analysis components into separate CPAs and reuse them in flexible combinations with other CPAs, instead of having to redefine them for every analysis from scratch.

For example, for most kinds of program analyses it is necessary to track the program counter, and it is often efficient to track the program counter explicitly rather than symbolically. Thus, we use the *Location CPA* $\mathbb{L}$ [40], which tracks exactly the program counter (with a flat lattice over all program locations, a constant precision, and the operators merge^{sep}, stop^{sep}, and prec^{id}), and we use this CPA in addition to other CPAs whenever explicit tracking of the program counter is necessary. The same can be done for other reusable analysis components, such as callstack tracking for an interprocedural analysis.

Furthermore, in order to track the abstract reachability graph (ARG) over the abstract states in the (flat) set reached, we define an additional *ARG CPA* $\mathbb{A}$, which stores the predecessor-successor relationship between abstract states. The ARG CPA allows us to reconstruct abstract paths in the ARG: An *abstract path* is a sequence $\langle e_0, \ldots, e_n \rangle$ of abstract states such that for any pair (e_i, e_{i+1}) with $i \in \{0, \ldots, n-1\}$ either e_{i+1} is an abstract successor of e_i, or e_{i+1} is the result of merging an abstract successor of e_i with some other abstract state(s). If both the Location CPA and the ARG CPA are used, we can reconstruct from an abstract path the path that it represents in the CFA.

3.3. Predicate Abstraction

Predicate abstraction [125] is a forward reachability analysis that unrolls the CFA into an ARG until a fixed point is reached. Abstract states are represented using predicates over program variables from a given set π (the *precision*). Given an abstract state e and a CFA edge with an operation op, we create a successor abstract state by computing a boolean combination of predicates from π that is an overapproximation of $\mathsf{SP}_{op}(e)$ (i.e., the concrete states reachable from e via *op*). This overapproximation is computed using either cartesian or boolean abstraction [12]. In order to speed up the coverage checks between abstract states (which are necessary for determining whether the fixed point was reached), binary decision diagrams (BDDs) [180] are used for representing the abstract states.

3.3.1. Counterexample-Guided Abstraction Refinement

Counterexample-guided abstraction refinement (CEGAR) [84] is an approach for iteratively finding an analysis precision that is strong enough to prove the program safe and coarse enough to allow for an efficient analysis. Starting with a coarse initial precision (typically an empty set of facts, e.g., predicates), an abstract model that is an overapproximation of the program is created by the underlying reachability analysis. If an abstract state that belongs to the error location is found in the abstract model, the concrete program path that leads to this state is reconstructed from the ARG and checked for feasibility. If the error path is feasible, the program is unsafe and the analysis terminates. Otherwise, the error path is infeasible, and we refine the precision of the analysis to be precise enough to eliminate this infeasible error path from the ARG. This is done by computing a Craig interpolant [139] for each location on the path and adding the predicates that are contained in these interpolants to the precision. Then the analysis is restarted. Due to the properties of interpolants, the precision of the reachability analysis will now be strong enough to allow ruling out the previously found infeasible error path. These steps are repeated until either a concrete error path is found, or the abstract model (and thus the program) is proven safe.

3.3.2. Lazy Abstraction

For improved performance, lazy abstraction [140] does not completely delete the previously computed ARG after the refinement step of the CEGAR algorithm, but recomputes only those parts of the ARG where the new predicates are necessary. Furthermore, the new predicates will not be used globally for all abstraction computations, but only in the part of the ARG and only at those locations of the CFA, for which they are relevant. The analysis terminates if either a feasible error path is found, or a fixed point is reached during unrolling the ARG (in which case the program is proven safe).

Predicate abstraction with CEGAR and lazy abstraction is for example implemented in the software model checker BLAST [35].

3.3.3. Adjustable-Block Encoding

Adjustable-block encoding (ABE) [42] aims at improving the performance of predicate abstraction by reducing the number of abstraction computations and refinements. It does not compute an abstraction for each new abstract state, but instead it groups abstract states into blocks and computes abstractions only once per block (at the end). With ABE, abstract states are now tuples of an *abstraction formula* and a concrete *path formula*. The path formula of any abstract state always represents a set of concrete paths from the block entry to the location of this abstract state. When a new abstract state is created, the strongest postcondition of the previous path formula and the current edge is created and used as the path formula of the new abstract state. The abstraction formula is copied from the previous abstract state. If there already exists an abstract state with the same location inside the same block, both states are merged into one abstract state by taking the disjunction of their path formulas. Only at the block end, an abstraction of the conjunction of the abstraction formula and the path formula of the current abstract state is computed and used as the new abstraction formula. The path formula is reset to *true* at the block end.

ABE does not only reduce the number of abstraction computations, but also the number of coverage checks (which are only done at block ends), and the size of the ARG (due to merging of abstract states). The latter is the reason for a vastly reduced number of refinements. During refinement, interpolants are computed only for abstract states at the block ends, because only for those abstract states predicates are needed for computing abstractions.

The block size for ABE can be freely chosen. If the block size is restricted to one single CFA edge (we name this *single-block encoding*, or SBE), an abstraction is computed for every new abstract state, corresponding to what is implemented for example in BLAST. Experiments have shown that for good performance, the program structure should be taken into account when defining the block encodings. A good configuration is for example to define block ends at loop-head locations of the program (ABE-Loops), such that the blocks will be the largest loop-free subgraphs of the CFA. Another suitable configuration with somewhat smaller blocks is to define block ends not only at loop heads but also at function entry and exit points (ABE-LF). This configuration of ABE has the same effect as using large-block encoding (LBE) [27], an alternative approach that also groups abstract states into blocks, but creates blocks of a fixed size during a preprocessing of the CFA instead of dynamically during the analysis like ABE.

Predicate abstraction with ABE is implemented, for example, in CPAchecker.

3.4. Impact (Lazy Abstraction with Interpolants)

The Impact algorithm [190] is another CEGAR-based approach that creates an unwinding of the CFA with abstract states that are labeled with a formula over the program variables. However, it never performs abstraction computations, and instead initializes all new abstract states to *true*. This is similar to how predicate-abstraction algorithms work while the precision is still empty.

The algorithm consists of three basic steps, which are applied until no further change can be made. In theory, the steps can be executed in any order, but the right strategy is crucial for good performance. The steps are:

Expand(e) If the abstract state e has no successors (i.e., it is a sink in the ARG) and is not covered, create the successor abstract states using *true* as their initial formula, and add them to the ARG.

Refine(e) If e is an abstract state at the error location with a formula different from *false*, compute inductive interpolants for the path from the ARG root to this abstract state. For each abstract state along this path, the corresponding interpolant is conjoined to the formula of the abstract state, and the abstract state is marked as not covered. If the path to the error location is infeasible, the formula of the abstract state at the error location is guaranteed to be *false* (which marks unreachable abstract states) after this step.

Cover(e_1, e_2) In this step, an abstract state e_1 is marked as *covered* by another abstract state e_2 if the following properties hold:

- e_2 is (and all of its ancestors are) not covered,
- both abstract states belong to the same program location,
- the formula of e_1 implies the one of e_2, and
- e_1 is not an ancestor of e_2.

If e_1 gets marked as covered, then (1) all abstract states that are covered by e_1 or e_1's children are uncovered, and (2) all children of e_1 are implicitly considered as covered. Note that covered abstract states never cover any other abstract states themselves, i.e., no chains of coverage exist. In order to prevent an infinite loop of coverings and uncoverings, the step *Cover* may be applied only to pairs (e_1, e_2) where e_1 was created after e_2 (only older abstract states can cover newer abstract states, not vice versa).

The application order of the steps as proposed by McMillan is to expand abstract states in a depth-first search. During the search, he keeps the invariant that the abstract state that is currently being expanded, and all its ancestors, are not coverable by any other abstract state (otherwise the current abstract state would not need to be expanded). As soon as an abstract state is found that belongs to the error location, the refinement procedure is run for this abstract state. After a successful refinement, the invariant that no abstract state on the path from the ARG root to the current abstract state is coverable is re-established by trying to cover all these abstract states. This can be optimized by checking only those abstract states that have been strengthened during refinement. This algorithm corresponds to the core algorithm of IMPACT, as presented by McMillan [190]. It is not available in IMPACT without the following optimization.

3.4.1. IMPACT with Forced Covering

When a new abstract state is created in the IMPACT algorithm, its formula is always *true* and thus it can only be covered by another abstract state at the same location that also has the formula *true*. However, after some refinements, most abstract states are expected to have stronger formulas, and thus coverage is unlikely, causing a large number of expansions and abstract states. As an optimization, one can try to immediately strengthen the formula of a new abstract state such that this abstract state can be covered by an existing abstract state at the same location. This is called *forced covering*. In order to forcefully cover an abstract state e_1 by another abstract state e_2, the path from the nearest common ancestor of both abstract states to e_1 is considered. If it can be proven that the formula of e_2 holds at the location of e_1 after following this path from the nearest common ancestor, the formula of e_2 can be set as the formula of e_1. Thus, e_1 is immediately covered by e_2. Additionally, the abstract states along the path from the nearest common ancestor to e_1 are strengthened by computing Craig interpolants for this path. This corresponds to the algorithm used for the benchmarks in the IMPACT article and to the tool implementation [190].

3.5. Bounded Model Checking and *k*-Induction

Bounded model checking (BMC) [54] is an approach that is aimed at finding bugs instead of proving safety. It unrolls all loops in the program up to a given loop bound k and encodes the resulting program in a single SAT or SMT formula, which is checked for satisfiability. The formula is constructed such that its satisfiability corresponds to whether a specification violation exists within the first k loop iterations of the programs. To check if we have already verified the full state space of the program or whether there are reachable states in the $k + 1^{\text{st}}$ loop iteration we can use a similar satisfiability check of the forward condition [120]. In the latter case we can retry the bounded model check with a higher value of k. However, iterating this process may not terminate if the program has unbounded loops.

In order to verify programs with unbounded loops, we can use an approach based on k-induction [111, 217]. Like a standard induction proof, a k-induction proof consists of two steps [229]: In the *base case*, we check that no specification violation exists within the first k loop iterations (this is the same as doing BMC with loop bound k). In the *step case*, we assume that no specification violation exists in any k consecutive loop iterations and check whether this implies that no specification violation exists in the $k + 1^{\text{st}}$ loop iteration. If both steps succeed, the program is safe. Otherwise we can iteratively try again using higher values of k (k-induction gets more powerful with increasing k [229]). Note that for software verification, it is usually necessary to use auxiliary invariants in addition to k-induction because k-induction alone is not strong enough to verify typical programs [110]. Besides using manually annotated invariants [111], auxiliary invariants for *k*-induction can be generated automatically using abstract interpretation [109, 207], or by deriving candidate invariants from templates and checking their validity using *k*-induction [64, 158]. It is beneficial to combine both kinds of invariant-generation techniques and to let the invariant generator(s) produce incrementally stronger invariants while running in parallel to the *k*-induction procedure [31].

A well-known bounded model checker is CBMC, and bounded model checking with *k*-induction is for example implemented in ESBMC.

3.6. CPAchecker

CPACHECKER[3] [41] is an open-source framework for software verification under the Apache 2.0 license based on the CPA concept. CPACHECKER supports verification of C programs (GNU C and C 11 [147]) using the parser of the Eclipse CDT project. Because it is written in Java it runs on all major platforms. CPACHECKER can also be used as a cloud service [34].

The first step of an analysis with CPACHECKER is the creation of a CFA for the given program. The core of CPACHECKER is the CPA algorithm (cf. Algorithm 3.1). This algorithm uses a CPA to analyze the given program. A typical configuration consists of some basic CPAs, e.g., the Location CPA for program-counter tracking (cf. Sect. 3.2.3), and one or more CPAs implementing the main abstract domain of the analysis, such as the Predicate CPA that we define in Chapter 4, all wrapped in a Composite CPA. Furthermore, the CPA algorithm can be wrapped inside further algorithms that implement approaches such as CEGAR (cf. Sect. 3.3.1), sequential combinations of analyses [100, 186, 231], or conditional model checking [36]. The verification result typically consists of the set of all reachable abstract states as determined by the CPA algorithm.

[3] `https://cpachecker.sosy-lab.org`

4. Predicate CPA

We define the core of our framework for predicate-based analyses as a CPA (with some extensions), which we name the *Predicate CPA* $\mathbb{P}$. It is an extension of an existing CPA for predicate abstraction with adjustable-block encoding (ABE) [42], and a preliminary version was already published [52].

The Predicate CPA $\mathbb{P} = (D_{\mathbb{P}}, \Pi_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \mathsf{merge}_{\mathbb{P}}, \mathsf{stop}_{\mathbb{P}}, \mathsf{prec}_{\mathbb{P}})$ consists of the abstract domain $D_{\mathbb{P}}$, the set $\Pi_{\mathbb{P}}$ of precisions, the transfer relation $\rightsquigarrow_{\mathbb{P}}$, the merge operator $\mathsf{merge}_{\mathbb{P}}$, the stop operator $\mathsf{stop}_{\mathbb{P}}$, and the operator $\mathsf{prec}_{\mathbb{P}}$ for dynamic precision adjustment. Additionally, we will define an operator $\mathsf{fcover}_{\mathbb{P}}$ for IMPACT-style forced covering and an operator $\mathsf{refine}_{\mathbb{P}}$ for refinements. In the following, we will define and describe these parts in more details. An overview over the architecture of the Predicate CPA can be seen in Fig. 4.1. In this UML-style diagram, dashed italic boxes represent components that have several possible implementations (shown below the dashed box) and can be configured as desired. In Chapter 5 we will then discuss how to use this CPA in various approaches for software model checking by choosing appropriate configurations.

4.1. Abstract Domain, Precisions, and CPA Operators

The abstract domain $D_{\mathbb{P}} = (C, \mathcal{E}_{\mathbb{P}}, [\![\cdot]\!]_{\mathbb{P}})$ consists of the set C of concrete states, the semilattice $\mathcal{E}_{\mathbb{P}}$ over abstract states, and the concretization function $[\![\cdot]\!]_{\mathbb{P}}$. The semilattice $\mathcal{E}_{\mathbb{P}} = (E_{\mathbb{P}}, \sqsubseteq_{\mathbb{P}})$ consists of the set $E_{\mathbb{P}}$ of abstract states and the partial order $\sqsubseteq_{\mathbb{P}}$.

4.1.1. Abstract States

Because of the use of ABE, an abstract state $e \in E_{\mathbb{P}}$ of the Predicate CPA is a triple $(\psi, l^{\psi}, \varphi)$ of an abstraction formula ψ, the abstraction location l^{ψ} (the program location where ψ was computed), and a path formula φ. Both formulas

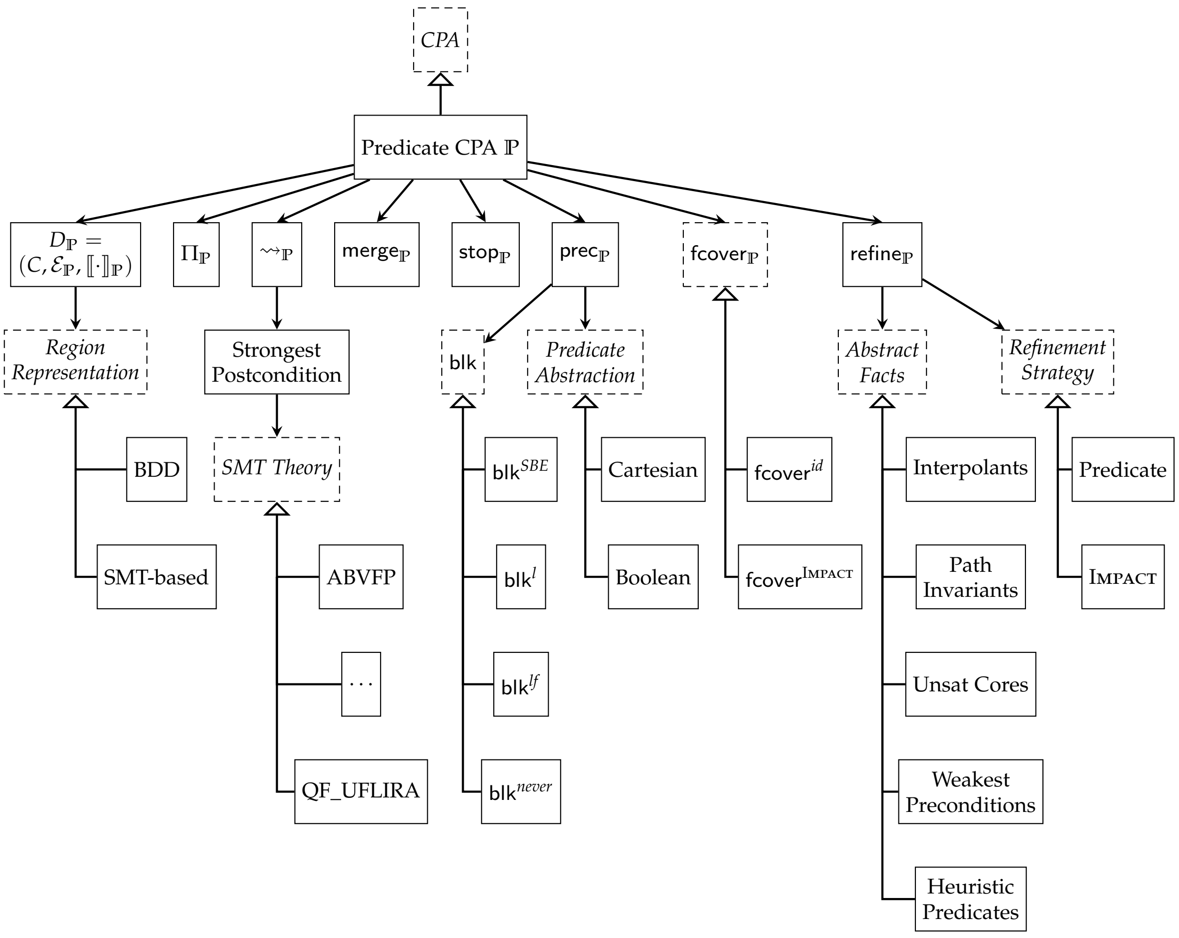


Figure 4.1.: Components of the Predicate CPA

are first-order formulas over predicates over the program variables from the set X, and an abstract state represents all concrete states that satisfy their conjunction: $[\![(\psi, l^{\psi}, \varphi)]\!]_{\mathbb{P}} = \{c \in C \mid c \models (\psi \wedge \varphi)\}$. The partial order $\sqsubseteq_{\mathbb{P}}$ is defined as $(\psi_1, l^{\psi}_1, \varphi_1) \sqsubseteq_{\mathbb{P}} (\psi_2, l^{\psi}_2, \varphi_2) = ((\psi_1 \wedge \varphi_1) \Rightarrow (\psi_2 \wedge \varphi_2))$, i.e., an abstract state is less than or equal to another state if it implies the other state. Abstract states where the path formula φ is *true* are called *abstraction states*, other abstract states are *intermediate states*. The transfer relation produces only intermediate states, and at the end of a block of program operations the operator prec computes an abstraction state from an intermediate state. The initial abstract state is the abstraction state $(true, l_{INIT}, true)$.

The path formula of an abstract state is always represented syntactically as an SMT formula. The representation of the abstraction formula, however, can be configured. We can either use a binary-decision diagram (BDD) [70], as in classic predicate abstraction [35, 125], or an SMT formula similar to the path formula. Using BDDs allows performing cheap entailment checks between abstraction states at the cost of an increased effort for constructing the BDDs.

4.1.2. Precisions

A precision $\pi \in \Pi_{\mathbb{P}}$ of the Predicate CPA is a mapping from program locations to sets of predicates over the program variables. This allows using a different abstraction level at each location in the program (lazy abstraction). The initial precision is typically the mapping $\pi(l) = \emptyset$, for all $l \in L$. However, one can also seed the analysis by supplying an initial precision that already contains some predicates, for example by using a heuristic that creates promising predicates from assume edges of the CFA, or by precision reuse [45] of the precision of a previous analysis of the same or a similar program. This can speed up the analysis substantially by reducing the amount of necessary refinements.

The standard Predicate CPA does not use dynamic precision adjustment [40] during an execution of the CPA algorithm: instead the precision is adjusted only during a refinement step, if the predicate refinement strategy is used. The only operation that changes its behavior based on the precision is the predicate abstraction that may be computed at block ends by the operator $\mathsf{prec}_{\mathbb{P}}$.

4.1.3. Transfer Relation

The transfer relation $(\psi, l^{\psi}, \varphi) \rightsquigarrow ((\psi, l^{\psi}, \varphi'), \pi)$ for a CFA edge (l_i, op_i, l_j) produces a successor state $(\psi, l^{\psi}, \varphi')$ such that the abstraction formula and location stay unchanged and the path formula φ' is created by applying the strongest-postcondition operator for the current CFA edge to the previous path formula: $\varphi' = \mathsf{SP}_{op_i}(\varphi)$. Note that this is an inexpensive, purely syntactical operation that does not involve any actual solving, and that it is a precise operation, i.e., it does not perform any form of abstraction. The details of how the semantics of program operations are encoded in the path formula are described in Sect. 4.4.

4.1.4. Merge Operator

The merge operator $\mathsf{merge}_{\mathbb{P}}$ combines intermediate states that belong to the same block (their abstraction formula and location is the same), and keeps any other abstract states separate:

$$\mathsf{merge}_{\mathbb{P}}((\psi_1, l^{\psi}_1, \varphi_1), (\psi_2, l^{\psi}_2, \varphi_2), \pi) = \\ \begin{cases} (\psi_2, l^{\psi}_2, \varphi_1 \vee \varphi_2) & \text{if } (\psi_1 = \psi_2) \wedge (l^{\psi}_1 = l^{\psi}_2) \\ (\psi_2, l^{\psi}_2, \varphi_2) & \text{otherwise} \end{cases}$$

This definition is common for analyses based on ABE. By merging abstract states inside each block, the number of abstract states in the ARG is kept small, and no precision is lost due to merging, because the path formula of an abstract state exactly represents the path(s) from the block start without abstraction. At the same time the loss of information that would lead to a path-insensitive analysis if states would be merged across blocks is avoided. The result is that the ARG, if projected to contain only abstraction states, forms an abstract reachability tree (ART) like in a path-sensitive analysis without ABE. This is necessary for being able to reconstruct abstract paths, for example during refinement and for reporting concrete error paths.

4.1.5. Stop Operator

The stop operator $\mathsf{stop}_{\mathbb{P}}$ checks coverage only for abstraction states, and always returns *false* for intermediate states:

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

Because the path formula of an abstraction state is always *true*, the first case is equivalent to checking if there exists an abstraction state $(\psi', \cdot, \mathit{true})$ in the set R whose abstraction formula ψ' is implied by the abstraction formula ψ of the current abstraction state $(\psi, l^{\psi}, \varphi)$. If abstraction formulas are represented by BDDs, this is an efficient operation, otherwise a potentially costly SMT query is required. The coverage check for intermediate states is omitted for efficiency, because it would always need to involve (potentially many) SMT queries. Note that this implies that infinitely long sequences of intermediate states must be avoided, otherwise the analysis would not terminate.

4.1.6. Precision-Adjustment Operator

The precision-adjustment operator $\mathsf{prec}_{\mathbb{P}}$ either returns the input abstract state and precision, or converts an intermediate state into an abstraction state performing predicate abstraction. The decision is made by the block-adjustment operator blk [42], which returns *true* or *false* depending on whether the current block ends at the current abstract state and thus an abstraction should be computed. The decision can be based on the current abstract state as well as on information about the current program location. We define the following common choices for blk: blk^{SBE} always returns *true*, leading to single-block encoding (SBE). blk^{lf} returns *true* at loop-heads, function calls/returns, and at the error location l_{ERR}, leading to a behavior similar to large-block encoding (LBE) [27]. blk^{l} returns *true* only at loop-heads and at the error location l_{ERR}. The abstraction at the error location is needed for detecting the reachability of abstract error states due to the satisfiability check that is implicitly done by the abstraction computation if the precision is not empty. blk^{never} always returns *false*. This will prevent all abstractions and (due to how $\mathsf{stop}_{\mathbb{P}}$ is defined) also prevents coverage between abstract states. This means that an analysis with blk^{never} will unroll the CFA endlessly until other reasons prevent this. We will show a meaningful application of blk^{never} in Sect. 5.2.1 (BMC).

A predicate abstraction $(\varphi)^{\rho}$ of a formula φ for a set ρ of predicates is a boolean combination of predicates from ρ that is at least as abstract as φ, i.e., $\varphi \Rightarrow (\varphi)^{\rho}$ for all sets ρ of predicates. To create an abstraction state from an intermediate state $(\psi, l^{\psi}, \varphi)$ at program location l (which is tracked by another CPA running in parallel and can be retrieved from there), we compute a predicate abstraction $(\psi \wedge \varphi)^{\pi(l)}$ for the formula $\psi \wedge \varphi$ and the set $\pi(l)$ of predicates from the precision, after adjusting the variable names of ψ to match those of φ (because the variables from ψ need to match the 'oldest' variables in φ). Thus, we can define the precision-adjustment operator as

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

Note that, if an abstraction is going to be computed, the current path formula φ precisely represents all the paths within this block (i.e., from the last abstraction state to the current abstract state). Thus, we name this path formula the *block formula* for the block ending in the current abstract state.

There exist two possible choices for computing the predicate abstraction: cartesian abstraction and boolean abstraction [12]. The *cartesian predicate abstraction* $(\varphi)^{\rho}_{\mathbb{C}}$ of a formula φ for a set ρ of predicates is the strongest *conjunction* of predicates. It can be computed by creating the conjunction of all predicates $p_i \in \rho$ which are implied by φ, i.e., $(\varphi)^{\rho}_{\mathbb{C}} = \bigwedge_{p \in \{p' \in \rho \mid \varphi \Rightarrow p'\}} p$. The *boolean predicate abstraction* $(\varphi)^{\rho}_{\mathbb{B}}$ of a formula φ for a set ρ of predicates is the strongest *boolean combination* of predicates. It can be computed using an SMT solver by solving $\varphi \wedge \bigwedge_{p_i \in \rho} (v_{p_i} \Leftrightarrow p_i)$ and enumerating all its models with respect to the fresh boolean variables $v_{p_1}, \ldots, v_{p_{|\rho|}}$ (this is called AllSMT). For each model we create a conjunction over the predicates from ρ, with each predicate p_i being negated if the model maps the corresponding variable v_{p_i} to *false*. The result of $(\varphi)^{\rho}_{\mathbb{B}}$ is the disjunction of all these conjunctions [177].

It has been shown that in practice boolean abstraction is too expensive for single-block encoding, whereas cartesian abstraction is too imprecise for larger blocks [27, 42], because with larger blocks disjunctions may appear in the path formula, and cartesian abstraction often overapproximates disjunctive facts too much. Note that if disjunctions appear for other reasons in the path formula, cartesian abstraction may be too imprecise even for single-block encoding. Optimizations are also possible, such as filtering out all predicates that are trivially irrelevant because their set of free variables does not intersect with the free variables of the input formula, or by computing the cartesian abstraction first and computing the boolean abstraction only over those predicates whose truth value could not be determined by the cartesian abstraction. If the precision is empty for the current program location, the outcome of the abstraction computation will always simply be *true* and no SMT queries are necessary. If the precision for the current program

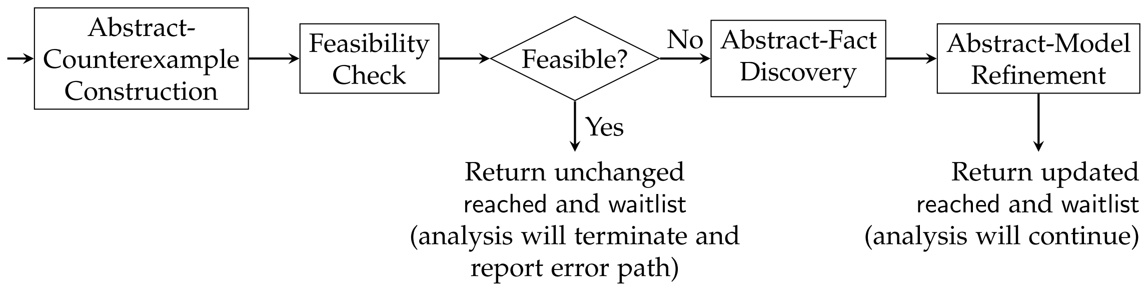

Abstract-Counterexample Construction
Feasibility Check
Feasible?
No
Yes
Abstract-Fact Discovery
Abstract-Model Refinement
Return unchanged reached and waitlist (analysis will terminate and report error path)
Return updated reached and waitlist (analysis will continue)

Figure 4.2.: Refinement steps

location is $\{false\}$, the abstraction computation will be equivalent to a simple satisfiability check, and the outcome will always be either *true* or *false*.

4.2. Refinement

The refinement operator $\mathsf{refine}_{\mathbb{P}}$ takes as input two sets $\mathsf{reached} \subseteq E \times \Pi$ of reached abstract states and $\mathsf{waitlist} \subseteq E \times \Pi$ of frontier abstract states, and expects $\mathsf{reached}$ to contain an abstract error state at error location l_{ERR} that represents a specification violation. refine either returns the sets unchanged (if the abstract error state is reachable, i.e., there is a feasible error path), or modified such that the sets can be used for continuing the state-space exploration with an increased precision (if the error path is infeasible). The operator works in four steps, which are shown in Fig. 4.2.

4.2.1. Abstract-Counterexample Construction

The first step is to construct the set of abstract paths between the initial abstract state and the abstract error state. Traditionally, in an abstract reachability tree, there would exist exactly one such abstract path. Because we use ABE, however, intermediate states can be merged, and thus the abstract states form an abstract reachability *graph*, where several paths can exist from the initial abstract state to the abstract error state. All these abstract paths to the abstract error state contain the same sequence of abstraction states with varying sequences of intermediate states in between. This is due to the fact that abstraction states are never merged, and intermediate states are merged only locally within a block. Thus, the ARG, if projected to the abstraction states, still forms a tree. The initial abstract state is always an abstraction state by definition, and our choices of the block-adjustment operator blk ensure that all abstract error states are also abstraction states. Thus, we define as *abstract counterexample* the sequence $\langle e_0, \ldots, e_n \rangle$ that begins with the initial abstract state ($e_0 = e_{INIT}$), ends with the abstract error state e_n, and contains all abstraction states $e_1, \ldots, e_{n-1}$ on paths between these two abstract states. This sequence can be reconstructed from the ARG by following a single arbitrary abstract path backwards from the abstract error state (using the information tracked by the ARG CPA), without needing to explicitly enumerate all (potentially exponentially many) abstract paths between the initial abstract state and the abstract error state.

4.2.2. Feasibility Check

From an abstract counterexample $\langle e_0, \ldots, e_n \rangle$ we can create a sequence $\langle \varphi_1, \ldots, \varphi_n \rangle$ of block formulas where each φ_i represents all paths between e_{i-1} and e_i. Note that each φ_i is also exactly the same formula as the path formula that was used as input when computing the abstraction for state e_i. Then we check whether there exists a feasible concrete path that is represented by one of the abstract paths of the abstract counterexample by checking the *counterexample formula* $\bigwedge_{i=1}^{n} \varphi_i$ for satisfiability in a single SMT query. If satisfiable, the analysis has found a violation of the specification and terminates. A concrete program path that is a witness for the specification violation [25, 30] can be reconstructed from a satisfying assignment for the counterexample formula. Otherwise, i.e., if all abstract paths to the abstract error state are infeasible under the concrete program semantics, we say that the abstract counterexample is spurious, and a refinement of the abstract model is necessary to eliminate this infeasible error path from the ARG.

4.2.3. Abstract Facts and Discovery Strategies

Afterwards, $\mathsf{refine}_{\mathbb{P}}$ discovers some abstract facts, which are later used for refining the abstract model. For an abstract counterexample $\langle e_0, \ldots, e_n \rangle$, a sequence $\langle \tau_0, \ldots, \tau_n \rangle$ of abstract facts can often be computed from the proof of unsatisfiability of the corresponding counterexample formula. Typical abstract facts about the analyzed program are interpolants or invariants, but other choices are also possible. We are interested in studying categories of abstract facts, and thus classify them according to the following properties:

(1) whether each fact in a sequence of abstract facts is guaranteed to hold at its respective location in the abstract counterexample, and whether the whole sequence is inductive ($\tau_0 = \mathit{true} \wedge \forall i \in [1, n] : (\tau_{i-1} \wedge \varphi_i \Rightarrow \tau_i)$);

(2) whether they guarantee progress by always ruling out the currently analyzed spurious abstract counterexample ($\tau_n = \mathit{false}$);

(3) whether they never contain irrelevant symbols;

(4) whether there exists an algorithm that is guaranteed to find such abstract facts; and

(5) whether they are always inductive invariants.

Note that no kind of abstract fact can fulfill all these properties, because finding inductive invariants that are strong enough to rule out a given infeasible path is in general undecidable.

Even abstract facts that are not guaranteed to hold or which do not guarantee progress may still be useful, for example when boolean combinations of these facts might fulfill these properties. Using abstract facts that do not fulfill property (3) can be inefficient, either because of the unnecessarily large formulas or because such abstract facts can reduce coverage between abstract states (if an abstract state is more precise than necessary). Inductive invariants are expected to be especially beneficial for the progress of the analysis, because these will not cause unrolling of loops, which can happen with noninductive abstract facts. Thus, it is expected that with inductive invariants the analysis may be able to converge faster.

Table 4.1 categorizes four kinds of abstract facts that are used by existing tools and approaches according to which of the presented properties hold for them. In the following, we describe them in more detail. Of course, it would also be possible to use a combination of various kinds of abstract facts, for example, falling back to something else if the primary choice fails or would use too many resources.

Table 4.1.: Classification of abstract facts according to fulfilled properties

	Property				
	(1)	(2)	(3)	(4)	(5)
Interpolants [139]	✓	✓	✓	✓	✗
Path Invariants [38, 174]	✓	✓	✗	✗	✓
Weakest Preconditions [140, 178, 183]	✗	✓	✗	✓	✗
Heuristic Predicates [104]	✗	✗	✗	✓	✗

Interpolants

The most commonly used kind of abstract facts are (inductive) Craig interpolants [97]. Given a sequence $\hat{\varphi} = \langle \varphi_1, \ldots, \varphi_n \rangle$ of formulas whose conjunction is unsatisfiable, a sequence $\langle \tau_0, \ldots, \tau_n \rangle$ is an inductive sequence of interpolants for $\hat{\varphi}$ if

1. $\tau_0 = \mathit{true}$ and $\tau_n = \mathit{false}$,
2. $\forall i \in \{1, \ldots, n\} : \tau_{i-1} \wedge \varphi_i \Rightarrow \tau_i$, and
3. for all $i \in \{1, \ldots, n-1\}$, τ_i references only variables that occur in $\bigwedge_{j=1}^{i} \varphi_i$ as well as in $\bigwedge_{j=i+1}^{n} \varphi_i$.

By definition, interpolants fulfill the properties (1), (2), and (3) of our categorization of abstract facts. For many common SMT theories, interpolants are guaranteed to exist and can be computed using off-the-shelf SMT solvers from a proof of unsatisfiability for $\bigwedge_{i=1}^{n} \varphi_i$, thus property (4) is typically fulfilled as well.

Note that in general there exist many possible sequences of interpolants for a single infeasible error path. Which interpolants are used can have a large effect on the performance of the whole analysis and even make the difference between finding a safety proof and not terminating at all [48], for example if the used interpolants lead to iterative unrolling of a long loop of the program. No known strategy exists to compute the best interpolant sequence for an abstract counterexample although several approaches for improving the quality of interpolants have been proposed (cf. Sect. 7.2).

Path Invariants

Another approach is to reconstruct a path program from the abstract counterexample and generate invariants for this path program [38]. A path program is created from the abstract counterexample by constructing a slice of the original program that consists of exactly the program locations and CFA edges that occur at least once in a path of the abstract counterexample. Intuitively, this means that any unwound loops in the abstract counterexample are replaced by the original loop. Thus, the path program represents not only all program paths that are present in the abstract counterexample, but also all other programs paths that traverse the same CFA edges with a different number of loop iterations. If there is an unbound loop in the program, a path program can represent infinitely many paths.

There exist several possibilities for computing invariants for a path program. One is to give the path program to an invariant generator [38] in the hope of retrieving invariants that are strong enough to prove infeasibility of the error location in the path program. Any kind of standard technique for invariant generation can be used for this, for example a constraint-based generator [38], or an abstract interpreter [127]. If the invariant generator succeeds, the produced *path invariants* can be used as abstract facts that fulfill the properties (1), (2), and (5). The advantage of this technique is that path invariants rule out not only the current spurious abstract counterexample, but also other (up to infinitely many) potential error paths, and prevent loop unrolling. The disadvantage is that the invariant generator is not guaranteed to succeed (property (4) does not hold), because path programs are full-fledged programs and can be as complex as the original program. However, the assumption is that path programs are typically only a small fragment of the original program, and the invariant generator is more likely to succeed on these smaller programs than on the original program.

Another possibility is to attempt to compute closed forms of the loops in the path program, and use these as invariants [174]. Computing closed forms of loops is called loop acceleration, and various techniques exist that are applicable to certain kinds of loops [14, 57, 61, 151, 173]. Loop acceleration will not always find a precise closed form for a given loop. In this case, it is also possible to combine loop acceleration with interpolation [72, 145], and compute interpolants for under- or overapproximations of the loop. Such interpolants can still be more general than interpolants that would be computed for the abstract counterexample itself.

Weakest Preconditions

The use of weakest preconditions [106] for abstraction refinement is an alternative to using interpolation, e.g., when suitable interpolating SMT solvers are not available for the required SMT theories. A weakest precondition can be seen as the inverse of the strongest postcondition that the operator $\mathsf{SP}(\cdot)$ computes: given a formula ψ that represents a region and a program operation *op*, the weakest precondition $\mathsf{wp}_{op}(\psi)$ represents the largest region such that applying *op* to each of its concrete states yields only concrete states represented by ψ. When we apply the weakest-precondition operator repeatedly for the operations along the spurious abstract counterexample, starting with the region *true* at the error location, we get an unsatisfiable formula (otherwise the abstract counterexample would be not be spurious). Thus, we can use the predicates that appear in the proof of unsatisfiability as abstract facts for refining the abstract model [140, 178]. This guarantees progress (property (2)) [140] and such predicates can always be computed if the underlying SMT theory is decidable (property (4)), however, contrary to interpolants, the predicates that are extracted from the proof of unsatisfiability are not guaranteed to hold at any location along the paths of the abstract counterexample, and they might contain irrelevant symbols. Of course, they are also not guaranteed to be invariants. This approach can be optimized by using unsat cores [82] in order to compute a subset of the clauses of the weakest preconditions that is sufficient for proving unsatisfiability [183]. This approach yields fewer irrelevant predicates, but still does not guarantee that only necessary symbols occur.

Heuristic Predicates

Instead of computing abstract facts that guarantee progress by ruling out the current spurious abstract counterexample, we can also apply some heuristics that generate predicates syntactically. While it might seem counterintuitive to use heuristics while other approaches that guarantee usable results exist, heuristics also have advantages. They are often cheap to compute, especially when compared to path invariants, and it may be possible to define heuristics that prefer abstract facts that are well-suited for the analysis (e.g., do not cause loop unrolling), whereas for example with interpolation it is more difficult to guide the computation towards “good” interpolants. In order to avoid running into a

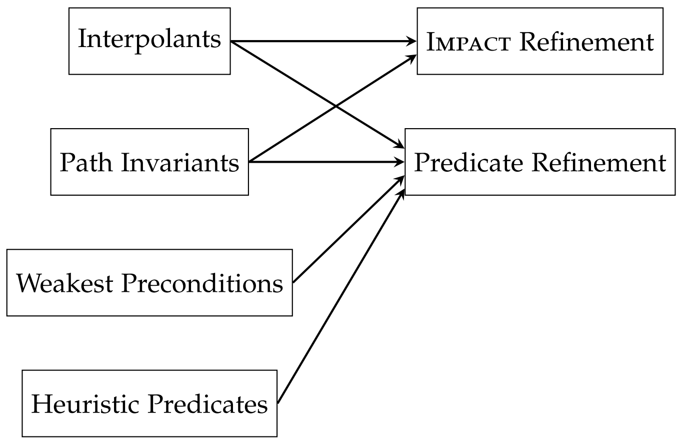


Figure 4.3.: Valid combinations of abstract facts and refinement strategies

loop that detects and refines the same abstract counterexample endlessly, we can use heuristics for predicate generation in a first attempt, and fall back to another approach when we encounter a previously seen abstract counterexample again.

Possible heuristics for generating predicates can for example include taking some of the conditions encountered along the error path. Specifically the conditions close to the error location can be relevant, because in cases like the verification of assertions that are included in the source code, these conditions encode the specification. It is also possible to look at the roles of variables [105] in the program and generate specific predicates depending on how each variable is used [104].

4.2.4. Refinement Strategies

Lastly, $\mathsf{refine}_{\mathbb{P}}$ needs to refine the precision of the analysis such that afterwards the analysis (hopefully) does not encounter the same error path again. A refinement strategy uses the current spurious abstract counterexample $\langle e_0, \ldots, e_n \rangle$ and the corresponding sequence $\langle \tau_0, \ldots, \tau_n \rangle$ of abstract facts to modify the sets $\mathsf{reached}$ and $\mathsf{waitlist}$. For this step, two common approaches exist. Afterwards, the refinement is finished, the modified sets $\mathsf{reached}$ and $\mathsf{waitlist}$ are returned to the analysis, and the analysis continues with building the abstract model (which will now be more precise). Figure 4.3 shows which refinement strategies can use which kinds of abstract facts.

IMPACT Refinement

One refinement strategy is to perform a refinement similar to the function REFINE of the IMPACT algorithm [190]. This strategy needs abstract facts that fulfill at least the properties (1) and (2), and is typically used with interpolants. The IMPACT refinement strategy takes each abstraction state e_i of the abstract counterexample, and conjoins to its abstraction formula the corresponding abstract fact τ_i (this would be unsound if property (1) does not hold). If an abstract state is actually strengthened by this (i.e., the previous abstraction formula did not already imply the abstract fact), we also need to recheck all coverage relations of this abstract state: If an abstract state e'_i previously covered by another abstract state e_i is now no longer covered, because the abstraction formula of e_i was strengthened by the refinement, we uncover and re-add all leaf abstract states in the subgraph of the ARG that starts with the uncovered abstract state e'_i to the set waitlist. We also check for each of the strengthened abstract states whether it is now covered by any other abstract state at the same program location. If this is successful, i.e., if a strengthened abstract state e_j is now covered by another abstract state e'_j, we mark the subgraph that starts with that strengthened abstract state e_j as covered and remove all leafs therein from waitlist (we do not need to expand covered abstract states). The only change to the set reached is the removal of all abstract states whose abstraction formula is now equivalent to *false* and their successors. Because the abstract facts fulfill property (2), this is guaranteed to be the case for at least the abstract error state.

Predicate Refinement

Another refinement strategy is used for traditional lazy predicate abstraction. It extracts predicates from the abstract facts, creates a new precision π with these predicates, and restarts (a part of) the analysis with a new precision that is extended by π. For each of these steps again some choices exist. This strategy can work with any kind of abstract fact, even those generated heuristically. Of course, progress is guaranteed only if the abstract facts fulfill the relevant properties.

For creating a set of predicates from an abstract fact τ, we can either take the singleton set $\{\tau\}$ or extract all basic atoms (i.e., the non-boolean parts of an SMT formula) that appear syntactically in τ. In the latter case, we can further try to filter out irrelevant predicates, for example those that are equivalent to another predicate. On the other hand, it can be worthwhile to derive and add additional predicates, for example adding the predicate $x \leq i$ for every predicate $x = i$ with a program variable x and an integer i. This may help in avoiding loop unrollings if the SMT solver produces interpolants with (overly specific) equalities where inequalities would be strong enough and the latter would be more general and useful to rule out other error paths than those of the current abstract counterexample as well.

Because a precision π is a mapping from program locations to sets of predicates, we can choose for which program location(s) new predicates should be added when creating the new precision. Assuming that, starting from an abstract counterexample $\langle e_0, \ldots, e_n \rangle$ with abstraction states at program locations $\langle l_0, \ldots, l_n \rangle$ we obtained a sequence $\langle \tau_0, \ldots, \tau_n \rangle$ of abstract facts and extracted a sequence $\langle \rho_0, \ldots, \rho_n \rangle$ of sets of predicates. Then we can add each predicate to the precision for the program location that corresponds to the point in the abstract counterexample where the predicate appears in the interpolant, i.e., $\pi(l) = \bigcup_{i=0}^{n}(\rho_i \text{ if } l = l_i \text{ else } \emptyset)$, or we can add all predicates globally (i.e., at all program locations) by using the precision $\pi(l) = \bigcup_{i=0}^{n} \rho_i$. Furthermore, we can also use other heuristics such as adding a predicate for all program locations that are in the same function as the program location for which the predicate has been found [35]. In any case, we take the precision π with the new predicates and the existing precision π_n that is associated in the set reached with the abstract error state e_n, and join them element-wise to create the new precision π' with $\forall l \in L : \pi'(l) = \pi_n(l) \cup \pi(l)$ that will be used in the subsequent analysis.

Finally, the sets reached and waitlist are prepared for continuing with the analysis. One possibility is to set both of them to $\{(e_0, \pi')\}$, i.e., remove all abstract states and restart the analysis at the initial state with the new precision. Lazy abstraction [140] is a more efficient alternative, where only those parts of the ARG are removed for which the new predicates are necessary. For this, we determine the first abstract state of the abstract counterexample for which the new precision π' would lead to more predicates being used in the abstraction computation than the originally used predicates, and call this the pivot abstract state. Then we remove the subgraph of the ARG that starts with the pivot abstract state from the sets reached and waitlist, as well as all abstract states that were covered by one of the removed abstract states. To ensure that the removed parts of the ARG get re-explored, we take all remaining parents of removed abstract states, replace the precision with which they are associated in reached with the new precision π', and add them to the set waitlist. This has not only the effect of avoiding the re-exploration of unchanged parts of the ARG, but also leads to the new predicates being used only in the relevant part of the ARG, with other parts of the program state space being explored with different (possibly more abstract and thus more efficient) precisions.

In general, the effect of the presented heuristics can be described as follows. Using exactly the discovered abstract facts (e.g., the interpolants) as predicates, for exactly their specific locations and only in the part of the ARG that needs to be rediscovered, is an approach that attempts to keep the precision of the analysis as abstract as possible and the abstract model of the program as coarse as possible. This minimizes the work for the abstraction computations during the analysis. However, this may lead to overly specific refinements that each rule out only a few infeasible error paths. Generalizing during refinement, e.g., by splitting abstract facts into atomic predicates, deriving additional predicates, or using the predicates at more program locations or in more parts of the state space, may help to rule out many more infeasible error paths with a single refinement, and thus might lead to faster convergence of the analysis.

Discussion

In comparison with the predicate-based refinement strategy, the Impact-style refinement strategy avoids the need for recomputing (parts of) the ARG. It also leads to a minimal increase in the precision of the analysis: the interpolants are used to strengthen abstract states only where absolutely necessary to rule out the current spurious abstract counterexample. There is no generalizing of the interpolants and no application of predicates to other parts of the program locations or state space. While both refinement strategies use lazy abstraction, using predicate refinement is typically more eager than Impact refinement: abstraction computations using the discovered predicates can generalize and derive more facts than what were present in the interpolants for the abstract counterexample, whereas with Impact refinement the abstract model will contain only exactly those facts that were discovered for some spurious abstract counterexample.

A further possibility is to combine both refinement strategies as it is done by the Ufo algorithm [2]: Primarily, the Impact refinement strategy is used.

Additionally, however, predicates derived from the interpolants can be added to the precision of abstract states like in the predicate refinement strategy, and these predicates will get used for abstraction computations when the ARG is further unrolled. This avoids the need for re-exploring existing parts of the ARG, which the predicate refinement strategy would cause, while making it possible to benefit from some of the generalization capabilities and the eagerness that the predicate refinement strategy provides.

4.3. Forced Covering

Forced coverings were introduced for lazy abstraction with interpolants [190] (Impact) for a faster convergence of the analysis. Typically, when the CPA algorithm creates a new successor abstract state for an Impact analysis, this new abstract state is too abstract to be covered by existing abstract states, since the Impact refinement strategy is used, which leads to all new abstraction states being equivalent to *true*. If an abstract state cannot be covered, the analysis needs to further create successors of it, leading to more abstract states and possibly more refinements. The idea of forced covering is to strengthen new abstract states such that they are covered by existing abstract states immediately if possible.

We define an operator $\mathsf{fcover}_{\mathbb{P}} : 2^{E \times \Pi} \times E \times \Pi \to 2^{E \times \Pi}$ that takes as input the set $\mathsf{reached}$ of reachable abstract states and an abstract state e with precision π, and returns an updated set $\mathsf{reached}'$ of reachable abstract states. The operator may replace e and other abstract states in $\mathsf{reached}$ with strengthened versions, if it can guarantee that this is sound and if afterwards the strengthened version of e is covered by another abstract state in $\mathsf{reached}'$. A trivial implementation of this operator is $\mathsf{fcover}^{id}(\mathsf{reached}, e, \pi) = \mathsf{reached}$, which does not strengthen abstract states and returns the set $\mathsf{reached}$ unchanged.

An alternative implementation is $\mathsf{fcover}^{\text{Impact}}$, which adopts the strategy for forced coverings presented for lazy abstraction with interpolants [190]. We extend this approach here to support adjustable-block encoding. Because the Predicate CPA does not attempt to cover intermediate states (only abstraction states), we also only attempt forced coverings for abstraction states. Given an abstraction state e that should be covered if possible, the candidate abstract states for covering are those abstraction states at the same location. For each candidate e', we first determine the nearest common ancestor abstraction state $\hat{e}$ of e and e' (using the information tracked by the ARG CPA). Now let us denote the abstraction formulas of e' and $\hat{e}$ with ψ' and $\hat{\psi}$, respectively, and let φ be the path formula that represents the paths from $\hat{e}$ to e. We then determine whether ψ' also holds for e by checking if $\hat{\psi} \wedge \varphi \Rightarrow \psi'$ holds, i.e., whether it is impossible to reach a concrete state that is not represented by ψ' when starting at $\hat{e}$ and following the paths to e. If this holds, we can strengthen the abstraction formula of e with ψ' (which immediately lets us cover e by e'). Furthermore, if there are abstraction states along the paths from $\hat{e}$ to e, we need to strengthen these states, too, in order to keep the ARG well-formed. We can do so by computing interpolants at the appropriate locations along the paths for the query that we have just solved, and strengthen the abstract states with the interpolants. If the query does not hold, we switch to the next candidate abstract state and try again. Finally, $\mathsf{fcover}^{\textsc{Impact}}$ returns an updated set $\mathsf{reached}$ with strengthened abstract states, or the original set $\mathsf{reached}$ if forced covering was unsuccessful for each of the candidates. Note that this forced-covering strategy is similar to interpolation-based refinement with the Impact refinement strategy, just that we attempt to prove that ψ' instead of *false* holds at the end of the path, and that the refined path does not start at the initial abstract state but at $\hat{e}$.

4.4. Encoding C Semantics

So far we have assumed that verification tasks are given in a simple iterative programming language with only integer variables and no heap memory. In this section we will now discuss how to model the semantics of the real-world programming language C. Most of the Predicate CPA is independent from the semantics of the analyzed programming language. We only need to replace the strongest-postcondition operator $\mathsf{SP}(\cdot)$ that is used by the transfer relation $\rightsquigarrow_{\mathbb{P}}$. Support for further procedural or object-oriented programming languages can be added in a similar way.

There exist already approaches of varying complexity and precision for encoding C semantics in first-order logic formulas for the purpose of automatic software verification, implemented in a number of tools. In the following, we summarize existing SMT-based approaches and discuss the variants that the Predicate CPA provides. We base our definitions on those from Sect. 3.1 and extend the possible variable types and operations. We keep the following restrictions:

1. The program must consist of correct C code (e.g., no type errors) and no behavior that is undefined according to the C standard may occur. Undefined behavior can lead to any consequence, thus, the verification of a program with undefined behavior would always have to report that the program is unsafe.
2. The program does not contain recursive function calls.
3. The program is not multithreaded.
4. The program must not use the C feature of nonlocal jumps, i.e., the macros `setjmp` and `longjmp`.

Furthermore, we assume that the program has been simplified syntactically as a preprocessing according to the following rules: All functions that are not defined by the C standard are inlined. Note that function calls through function pointers can still be supported by priorly replacing them with regular function calls to all possible candidate functions, each with a guard that compares the value of the function pointer with the address of the respective candidate function. Variables are renamed such that their names are globally unique. All declarations of variables and types are omitted, and variable initializers are kept as assignments. Expressions are free of side effects (possible by introducing auxiliary program variables and operations) and use only a minimal set of necessary operators [88].

Then we can represent C programs as CFAs where the operations attached to the edges can have exactly one of the following forms:

- an assumption $[p]$,
- a simple assignment $x := e$,
- a function call $func(e^*)$, and
- a function-call assignment $x := func(e^*)$.

Here, e is an expression, p is a predicate, $x \in X$ is a program variable, and $func(e^*)$ is a call to a function from the C standard library with an arbitrary amount of parameter expressions. Program variables in X can have any of the types defined by the C programming language. Expressions and predicates refer only to program variables from X, and contain only the following operators:

- the arithmetic operators $+$, $-$, $*$, $/$, $\%$,
- the bitwise operators $\ll$, $\gg$, &, $|$,^,
- casts,
- the address-of and indirection operators & and $*$, and
- the field-access operator $.f$ for structs and unions.

In theory, the strongest-postcondition operator $\mathsf{SP}_{op}(\cdot)$ needs to precisely encode the semantics of op as a formula. However, there are two features in C that may create problems in practice when being modeled precisely: nonlinear arithmetic (including bitwise operators, overflows, and all floating-point arithmetic) and unbounded heap usage. However, not in all use cases these features are important. For example, for control-flow heavy programs like those encoding a finite automaton using some integer state variables, there often exist interesting properties to be verified that do not depend on nonlinear arithmetic and heap usage. Thus, some users might be willing to sacrifice precision or even soundness for C programs that rely on these two features in order to gain a more effective and efficient verification procedure, and this is implemented in several commonly used tools [13, 35]. In our Predicate CPA this can be achieved by using a strongest-postcondition operator that only approximates the semantics of the program. Note that this approximation is different from the abstraction done by the operator $\mathsf{prec}_{\mathbb{P}}$ and independent from the precisions in $\Pi_{\mathbb{P}}$. A precision $\pi \in \Pi_{\mathbb{P}}$ can never make the analysis more precise than the approximation used by $\mathsf{SP}_{op}(\cdot)$.

In order to create a flexible verification framework that suits a wide range of use cases we provide several different variants of $\mathsf{SP}_{op}(\cdot)$, which differ in precision and soundness and are characterized by the SMT theories they require. In total, there are eight different combinations of SMT theories for encoding (approximate) C semantics. Table 4.2 shows all these combinations and indicates the most important restrictions relevant for each. Note that none of these combinations is free from restrictions. In the following, we discuss the possible choices to be made for selecting an appropriate strongest-postcondition operator in more detail.

Table 4.2.: Possible combinations of SMT theories for encoding program semantics (with names according to the naming scheme for SMT-LIB logics [17])

	Quantifier		Quantifier-free	
	Array	UF	Array	UF
Bitprecise	ABVFP [a]	UFBVFP [a]	QF_ABVFP [b]	QF_UFBVP [c]
Linear	AUFLIRA [ad]	UFLIRA [ad]	QF_AUFLIRA [bd]	QF_UFLIRA [cd]

[a] undecidable
[b] incomplete interpolation (interpolants may contain quantifiers)
[c] bounded heap
[d] imprecise or unsound nonlinear arithmetic

4.4.1. Nonlinear Arithmetic

First, we need to choose the SMT theories for encoding arithmetic expressions, i.e., which row we use in Table 4.2. Using the theory of fixed-width bitvectors (BV) [18, 98] for integer types and the floating-point theory (FP) [66, 134, 212] for floating types will allow a precise encoding of most arithmetic C expressions [9, 96]. Both theories allow to encode program variables with their actual size in bits and model overflows and casts appropriately. The arithmetic and bitwise operators of C can be encoded using the respective operators of the two theories. The theory of bitvectors is for example used by tools like Esbmc [96] and Llbmc [116]. Other tools achieve a similar precision by encoding program semantics in propositional logic and using a SAT solver [86, 87, 94, 148].

However, formulas over the bitvector and floating-point theories can be expensive to solve, and not all SMT solvers support them (cf. Sect. 6.3). Interpolation over bitvectors exists [128], but in practice support for it is still sometimes problematic as we will see in our evaluation in Sect. 16.1. Thus, it can be necessary to make a trade-off and use the theory of linear arithmetic over integers or reals in combination with the theory of uninterpreted functions for approximating nonlinear program behavior (the second row in Table 4.2).

The approximation of bitvectors with linear arithmetic over integers and uninterpreted functions can be sound (but still imprecise), if we ensure that every satisfying assignment of a formula that encodes a program statement with bitvectors is also a valid satisfying assignment for the formula that approximates the same program statement with integers. For this we have to specifically handle all operators that have nonlinear semantics and ensure that, whenever the operation is nonlinear, the result of the operation in the formula is a nondeterministic value. For example, when two program variables are multiplied, we can approximate this by encoding the multiplication as an uninterpreted function. Using an uninterpreted function instead of setting the result to a fresh variable is stronger because it allows the solver to deduct that the result of two multiplications with equal inputs will be the same, even though it cannot infer the concrete value. Nonlinear behavior of bitvector operations also occurs for example when a result overflows. In the following example, we denote bitvector variables and operations with an index that specifies their bit-width. Variables and operations without index are of type integer. The unsigned bitvector addition $x_w +_w y_w$ of two bitvectors x_w and y_w with equal width w can be approximated with $ite(x + y < 2^w, x + y, cast^w(x + y))$, where ite is the ternary operator, and $cast^w$ is an uninterpreted function that we use for overflowing results of width w. Note that the value of w is known during the creation of the formula and thus the value of 2^w can be calculated directly, no nonlinear operation is necessary in the formula. Because we assume that no undefined behavior occurs in the program, we do not need to handle signed overflows, which are undefined behavior. However, for verifying real-world software it might be more practical to approximate signed overflows as if two's complement is used [1], which can be done similarly to unsigned overflows. All other bitvector operators can be handled in the same way. Some tools even use an encoding for integer variables that approximates bitvector operators directly with their linear equivalent (omitting the disjunction), and are thus unsound in presence of overflows, e.g., Blast [35].

For the approximation of floating-point arithmetic, some tools use linear arithmetic over reals [15, 96, 194], or implement fixed-point arithmetic using bitvectors [95]. Both approximations are of course imprecise and unsound [124], but can be a pragmatic way for handling programs in which floating-point variables may appear from time to time, but are not crucial for the safety of the program. This can for example be the case in control-flow heavy programs that implement finite automata or low-level systems programs, both of which are a common use case for software verification. There also exist possibilities for sound approximation

[1] The C standard does not require a specific encoding of signed integers, but most hardware architectures use two's complement, and some programmers rely on this.

of floating-point arithmetic, e.g., using projection functions over floating-point intervals [60], expression canonization [92], or abstraction [66]. However, in cases where the semantics of floating-point arithmetic are not important for the verification, we might as well choose a simpler and faster approximation, and in cases where a higher precision is important, we expect that in practice any approximation might not be enough and a fully precise encoding needs to be adopted anyway. Thus, we decide to model floating-point arithmetic either precisely or by a coarse approximation using linear arithmetic over reals and uninterpreted functions. Furthermore, this allows us to use off-the-shelf SMT solvers without much additional implementation effort.

4.4.2. Pointer Accesses

Second, we need to decide whether to use a decidable encoding for pointer accesses that is only able to handle bounded sizes of memory regions, or an undecidable encoding that allows unbounded memory regions. The bounded encoding can be done using the theory of uninterpreted functions: a unary function symbol (e.g., m) is used to represent program memory, and a read through a pointer is encoded by applying the function to the address stored in the pointer. A write through a pointer is encoded by creating a fresh function symbol m' (assuming we use the skolemized encoding from Sect. 3.1) and adding constraints that m' applied to the address stored in the pointer is equal to the value that is written there, and for all other addresses the result of m' is equal to the result of m. Thus, the operation `*p = e;` is encoded by $m'(p) = e \wedge \forall i \in M : i \neq p \Rightarrow m'(i) = m(i)$. Here, the range M represents the finite bounds of memory regions, and thus the universal quantifier can be unrolled, resulting in a decidable (though potentially large) formula.

An unbounded encoding of pointers can be achieved with the theory of uninterpreted functions plus quantifiers by simply removing the bound M from the encoding of write operations. However, the combination of uninterpreted functions and quantifiers is undecidable [58]. Instead of quantifiers we could use the SMT theory of arrays [189], where a read through a pointer can be encoded by the theory operation `select`, and a write can be encoded by the theory operation `store` [139]. This theory is decidable [221], however, quantifiers may still be necessary even with the theory of arrays because interpolation for arrays is not

```
(declare-fun m1 () (Array (_ BitVec 32) (_ BitVec 8)))
(declare-fun m2 () (Array (_ BitVec 32) (_ BitVec 8)))
(declare-fun p () (_ BitVec 32))
(declare-fun q () (_ BitVec 32))
(assert
  (let ((v1 (concat (select m1 (bvadd p #x00000003))
                    (select m1 (bvadd p #x00000002))
                    (select m1 (bvadd p #x00000001))
                    (select m1 p)))
        (v2 (concat (select m1 (bvadd q #x00000003))
                    (select m1 (bvadd q #x00000002))
                    (select m1 (bvadd q #x00000001))
                    (select m1 q))))
    (let ((v3 (bvadd v1 v2)))
      (= m2
        (store
          (store
            (store
              (store m1 (bvadd p #x00000003) ((_ extract 31 24) v3))
              (bvadd p #x00000002) ((_ extract 23 16) v3))
            (bvadd p #x00000001) ((_ extract 15 8) v3))
          p ((_ extract 7 0) v3))))))
```

Figure 4.4.: Statement `*p = *p + *q` encoded in SMTLIB with a byte-wise memory array; input and output array are named `m1` and `m2`, respectively.

guaranteed to produce quantifier-free interpolants [161]. Interpolating solvers with support for quantifiers and arrays exist [191], but the combination of arrays and quantifiers is again undecidable in general [63, 221]. Note that variants of the array theory may guarantee quantifier-free interpolants [69, 154, 155].

4.4.3. Heap Memory

Third, we need to decide how to model the program's heap. This does not change which SMT theories are used, but only how accesses to heap memory are encoded. One possibility is to model the whole program heap as one large contiguous array of bytes [219]. This encoding resembles closely how today's machines organize the memory and allows handling out-of-bounds pointer accesses that overflow into adjacent memory regions, casts between different pointer types (e.g., from pointer-to-int to pointer-to-float), and misaligned pointer accesses (e.g., a read of a value from address $n + 2$ after a four-byte value has been written to address n). However, most of these operations are undefined according to the C standard, and this encoding produces complex formulas that may be expensive to solve, for example because of the repeated splitting and reassembling of values into bytes. Consider as an example the formula from Fig. 4.4, which encodes the statement `*p = *p + *q` by reading the eight bytes for `*p` and `*q` individually from the memory (storing them in the variables `v1` and `v2`), adding these values (storing the result in the variable `v3`), and finally storing the four bytes individually in the array that is used for representing heap memory. Furthermore, this byte-wise encoding is not appropriate if we use theories with linear arithmetic for approximating arithmetic operations.

In order to avoid this, we can assume independent heaps for each variable type, and use a separate array for each such type. This is known as the Burstall-Bornat model [59, 71] or as typed memory [89], and implemented for example in Blast [35]. With this memory model, we would use one array for 32-bit ints, one for 64-bit ints, one for floats etc., and each of these arrays has the element type that we also use for arithmetic operations on these variable types. Which array is used for a pointer access is defined by the type of the pointer that is being accessed. The encoding is now easier and more efficient for the solver, but pointer casts and misaligned accesses will now no longer resemble a real machine, but produce nondeterministic values. Even finer-grained partitionings of the heap are possible, for example by using a separate array for each field of each struct type [59, 228]. However, this fails for example if pointer arithmetic is used to access fields.

In any case of these cases, it is necessary to add constraints whenever memory is allocated with `malloc` that ensure that the newly allocated memory does not overlap with any previously allocated memory [219].

These memory encodings are possible regardless of whether we use the theory of arrays or uninterpreted functions for pointer accesses. Possibly aliased variables on the program stack need to be encoded in the same way as memory on the heap, but variables that are guaranteed to have no aliases can be encoded directly (i.e., without indirection) as an optimization.

An alternative to encoding program memory as one or more arrays is to represent it as a set of "objects", which each object being one (dynamically allocated) memory region [96]. In this case, pointers consist of a pair of an identifier for the target object and the offset within this object. This encoding is for example used by Cbmc and Esbmc.

Note that because we have assumed that no undefined behavior occurs, we did not discuss adding checks for conditions such as that all memory accesses occur within the correct bounds and that memory is never freed twice. If desired, such checks can be implemented for any of the presented heap encodings [96, 219].

5. Applications

In this chapter, we will present and discuss several applications of our framework for predicate-based software verification. We will use the Predicate CPA to unify four well-known verification approaches in a common presentation that highlights their main differences. This enables us to study the differences of these approaches and understand their core concepts. Furthermore, we will present extensions and new variants of the existing approaches that are made possible by having expressed the approaches in our unifying framework. The applications of this chapter show the advantage of having a flexible and configurable framework for unifying approaches for predicate-based software verification.

5.1. An Extended CPA Algorithm

In order to be able to use all the features of the Predicate CPA and support approaches such as lazy abstraction, we first need to slightly extend the CPA algorithm. The extended version, which we call the CPA++ algorithm, is shown as Algorithm 5.1. Compared to the original version (Algorithm 3.1), it has the following differences:

1. CPA++ gets reached and waitlist as input and returns updated versions of both of them, instead of getting an initial abstract state and returning a set of reachable abstract states.
2. CPA++ calls a function abort to determine whether it should abort early for each found abstract state (lines 16 to 17).
3. CPA++ calls the precision-adjustment operator immediately for each new abstract state (line 7) instead of only before expanding an abstract state.
4. CPA++ attempts a forced covering by calling fcover before expanding an abstract state (lines 3 to 5).

Algorithm 5.1 CPA++($\mathbb{D}$, reached, waitlist, abort), extension of Algorithm 3.1

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

```
1:  while waitlist $\neq \emptyset$ do
2:    pop $(e, \pi)$ from waitlist
3:    reached := fcover(reached, $e, \pi$)
4:    if $(e, \pi) \notin$ reached then
5:      continue                                   // Forced covering was successful.
6:    for all $e'$ with $e \rightsquigarrow (e', \pi)$ do
7:      $(\hat{e}, \hat{\pi})$ := prec($e', \pi$, reached)        // Adjust the precision.
8:      for all $(e'', \pi'') \in$ reached do
9:        $e_{new}$ := merge($\hat{e}, e'', \hat{\pi}$)      // Combine with existing abstract state.
10:       if $e_{new} \neq e''$ then
11:         waitlist := $\big($waitlist $\cup \{(e_{new}, \hat{\pi})\}\big) \setminus \{(e'', \pi'')\}$;
12:         reached := $\big($reached $\cup \{(e_{new}, \hat{\pi})\}\big) \setminus \{(e'', \pi'')\}$;
13:     if not stop($\hat{e}, \{e \mid (e, \cdot) \in$ reached$\}, \hat{\pi}$) then   // Add new abstract state?
14:       waitlist := waitlist $\cup \{(\hat{e}, \hat{\pi})\}$
15:       reached := reached $\cup \{(\hat{e}, \hat{\pi})\}$
16:       if abort($\hat{e}$) then
17:         return (reached, waitlist)
18: return (reached, waitlist)
```

The first two changes allow calling CPA++ iteratively and keep expanding the same set of abstract states, which is necessary for CEGAR with lazy abstraction (where we want to abort as soon as we find an abstract error state, and continue after refinement without restarting from scratch; abort is then implemented to return *true* for abstract error states). The new position of the call to the precision-adjustment operator is necessary because previously the resulting abstract states ($\hat{e}$ in Algorithm 3.1) were never put into reached. However, we need the abstract states resulting from prec to be in reached, because among them are the abstraction states of the Predicate CPA, which are necessary for refinement (cf. Sect. 4.2.1).

Similar changes to the CPA algorithm have been used previously [44, 52]; we now combine them in order to provide an all-encompassing algorithm for reachability that we can use as building block for our unifying framework for predicate-based software verification.

5.2. Unifying SMT-Based Approaches for Software Verification

The presented Predicate CPA is the base of a unifying framework for predicate-based software verification, and it is possible to express several existing approaches within this framework. Four such well-known approaches are bounded model checking (BMC), *k*-induction, predicate abstraction, and the Impact algorithm.

We can express these four approaches by taking an appropriately configured instance of the Predicate CPA $\mathbb{P}$ from Chapter 4, combine it with the typical basic CPAs such as the Location CPA $\mathbb{L}$ and ARG CPA $\mathbb{A}$ from Sect. 3.2.3 in a Composite CPA (cf. Sect. 3.2.2), and create a reachability analysis using the CPA++ algorithm with this combination of CPAs. Additionally, we may wrap the CPA++ algorithm inside another algorithm that implements further approaches upon the basic reachability analysis. Table 5.1 summarizes which components (as shown in Fig. 4.1) of our Predicate CPA need to be chosen for each of the four mentioned approaches, and in the following, we discuss each configuration. Note that the choice of the strongest-postcondition operator is in principle orthogonal to the choice of the approach we want to express, and thus, we can combine each approach with all the differently powerful encodings of the program semantics. However, if using interpolation for refinement the restrictions about incomplete interpolation for quantifier-free SMT theories with arrays apply.

5.2.1. Bounded Model Checking

For bounded model checking [54], we set the ABE block size to infinite (we call this whole-program encoding) by using the block operator blk^{never}, and we use fcover^{id} (i.e., no forced coverings). Additionally, we combine the Predicate CPA with a CPA for bounding the state space besides the typical basic CPAs.

The *Loop-Bound CPA* $\mathbb{LB}$ tracks in its abstract states for every loop of the program how often the loop body was traversed on the current program path. It associates each loop-head location with a counter that starts with -1 and is incremented by the transfer relation whenever the respective location is reached. The precision is the loop bound k: $\pi = k$, with $k > 0$. The transfer relation of the Loop-Bound CPA is unsound by purpose: it does not produce any successor

Table 5.1.: Configurations of Predicate CPA for common SMT-based approaches (cf. Fig. 4.1 for components; — represents unused components)

	BMC & k-Induction	Predicate Abstraction	IMPACT
Region Representation	—	BDD	SMT-based
Strongest Postcondition	any	any	any
blk	blk^{never}	e.g.: blk^{lf}, blk^{l}	blk^{SBE}
Predicate Abstraction	—	any	—
fcover	fcover^{id}	fcover^{id}	any
Refinement:			
– Abstract Facts	—	any	Interpolants
– Refinement Strategy	—	Predicate	IMPACT

abstract states for abstract states in which one of the counters for the loop-head locations is equal to the loop bound k in the precision, and thus prevents the analysis from exploring any paths for more than k loop iterations. Apart from that, the Loop-Bound CPA uses the standard operators merge^{sep}, stop^{sep}, and prec^{id}.

This configuration leads to an analysis without abstraction computations, expensive coverage checks, and refinements. Instead, the CPA++ algorithm simply unrolls the CFA (within the loop bound), and each abstract state contains a path formula that exactly represents the paths from the initial location to this abstract state. We wrap the CPA++ algorithm in another algorithm that checks satisfiability of the path formula of each abstract error state after the CPA++ algorithm has finished (we can use Algorithm 5.2, which is discussed in Sect. 5.2.2, for this by omitting lines 15 to 23). If at least one path formula is satisfiable (for efficiency, we check the disjunction of all path formulas at once in line 10 of Algorithm 5.2), then there exists a feasible path to the error location, i.e., the specification is violated.

We can also implement a forward-condition check [120] by making an additional SMT query for the satisfiability of the path formulas of all those abstract states for which the Loop-Bound CPA has unsoundly restricted the successor abstract states. If none of these path formulas is satisfiable, the specification is proven to hold for the program. If for a given loop bound k the result was inconclusive (i.e., no specification violation found but the forward-condition check was unsuccessful, too), we can repeat the bounded model check with a higher k.

5.2.2. *k*-Induction

For a k-induction-based verification approach [111, 217] we can use the same configuration of the Predicate CPA with blk^{never} as for bounded model checking. For ease of presentation, we assume here that the loop head is not reachable from the error location l_{ERR} and that the analyzed program has exactly one loop whose loop-head location is l_{LH}. In practice, k-induction can be applied to programs with many loops [31].

We present an algorithm for k-induction-based verification based on the Predicate CPA as Algorithm 5.2. This algorithm supports iterative deepening and injection of continuously refined invariants. We can use this algorithm in combination with (external) standard invariant-generation techniques, such as data-flow analysis [167, 200] and template-based approaches [37, 93]. This is necessary, because often the safety property of a verification task is not directly k-inductive for any k, but only relative to some auxiliary invariant, so that plain k-induction cannot succeed in proving safety. Strengthening the hypothesis of the inductive-step case with auxiliary invariants may allow the algorithm to prove such properties as well.

Algorithm 5.2 gets as input initial and maximal values for the loop bound and a function that computes the next loop bound after each iteration (this function can for example increase the value by one, or double it). Additionally we give the algorithm a combination of CPAs (as a composite CPA) that includes the Location CPA $\mathbb{L}$ (cf. Sect. 3.2.3), our Predicate CPA $\mathbb{P}$ in the configuration for bounded model checking, and the Loop-Bound CPA $\mathbb{LB}$ (cf. Sect. 5.2.1). Thus, each abstract state is a tuple of the current program counter l (this is an abstract state of $\mathbb{L}$), a predicate abstract state (which is itself a tuple of an abstraction formula, an abstraction location, and a path formula), and a mapping of loop heads to loop counters (this is an abstract state of $\mathbb{LB}$).

Algorithm 5.2 Iterative-Deepening k-Induction with Invariants (adapted from [31])

Input:
the initial value $k_{init} \geq 1$ for the bound k,
an upper limit k_{max} for the bound k,
a function $\mathsf{inc} : \mathbb{N} \to \mathbb{N}$ with $\forall n \in \mathbb{N} : \mathsf{inc}(n) > n$ for increasing the bound k,
a composite CPA $\mathbb{D}$ with the Location CPA $\mathbb{L}$, the Predicate CPA $\mathbb{P}$, and the Loop-Bound CPA $\mathbb{LB}$ as components,
for which E denotes the set of composite abstract states and Π the set of precisions
Output: **false** if l_{ERR} is reachable, **true** otherwise
Variables: the current loop bound $k \in \mathbb{N}$,
two abstract states $e_{INIT} \in E$ and $e_{LH} \in E$ and a precision $\pi_{INIT} \in \Pi$,
two sets reached and waitlist of elements of $E \times \Pi$, and
a function $\mathsf{abort}^{never} : E \to \mathbb{B}$

```
1:  $k := k_{init}$
2:  $e_{INIT} := (l_{INIT}, (true, l_{INIT}, true), \{l_{LH} \mapsto -1\})$   // Create initial abstract state.
3:  $e_{LH} := (l_{LH}, (true, l_{LH}, true), \{l_{LH} \mapsto 0\})$   // Create abstract state at $l_{LH}$.
4:  $\mathsf{abort}^{never} := \{\cdot \mapsto false\}$   // $\mathsf{abort}^{never}$ always returns $false$.
5:  while $k \leq k_{max}$ do
6:    $\pi_{INIT} := \{(\emptyset, \{\cdot \mapsto \emptyset\}, k)\}$   // Create initial precision.
7:    $\mathsf{reached} := \mathsf{waitlist} := \{(e_{INIT}, \pi_{INIT})\}$
8:    $(\mathsf{reached}, \mathsf{waitlist}) := \mathsf{CPA{+}{+}}(\mathbb{D}, \mathsf{reached}, \mathsf{waitlist}, \mathsf{abort}^{never})$
9:    $base\_case := \bigvee \{\varphi \mid ((l_{ERR}, (\cdot, \cdot, \varphi), \cdot), \cdot) \in \mathsf{reached}\}$
10:   if $\mathsf{sat}(base\_case)$ then
11:     return false
12:   $forward\_condition := \bigvee \{\varphi \mid ((l_{LH}, (\cdot, \cdot, \varphi), i), \cdot) \in \mathsf{reached} \land i(l_{LH}) = k\}$
13:   if $\neg\,\mathsf{sat}(forward\_condition)$ then
14:     return true
15:   $\pi_{INIT} := \{(\emptyset, \{\cdot \mapsto \emptyset\}, k+1)\}$   // Precision with loop bound $k+1$.
16:   $\mathsf{reached} := \mathsf{waitlist} := \{(e_{LH}, \pi_{INIT})\}$
17:   $\mathsf{reached} := \mathsf{CPA{+}{+}}(\mathbb{D}, \mathsf{reached}, \mathsf{waitlist}, \mathsf{abort}^{never})$
18:   $step\_case := \bigvee \{\varphi \mid ((l_{ERR}, (\cdot, \cdot, \varphi), i), \cdot) \in \mathsf{reached} \land i(l_{LH}) = k\}$
19:   repeat
20:     $Inv := \mathsf{get\_currently\_known\_invariant}()$
21:     if $\neg\,\mathsf{sat}(Inv \land step\_case)$ then
22:       return true
23:   until $Inv = \mathsf{get\_currently\_known\_invariant}()$
24:   $k := \mathsf{inc}(k)$
25: return unknown
```

For each value of the loop bound k as determined by the initial and maximal values and the increment function, the algorithm performs the checks for base case, forward condition, and step case. For the *base case* (lines 6 to 11), which is identical to bounded model checking, we set the bound of the Loop-Bound CPA to k and use the CPA++ algorithm (Algorithm 5.1) to unroll the program with an abstract state e_{INIT} at the initial program location as initial abstract state and the precision π_{INIT} as initial precision (the Location CPA has an empty precision, the Predicate CPA has a precision that maps all program locations to an empty set of predicates, and the Loop-Bound CPA has a precision that consists of the single constant value k). Then we create a disjunction of the path formulas of all resulting abstract states at the error location. Because of the configuration of the Predicate CPA and the Loop-Bound CPA, this formula represents all paths from l_{INIT} to l_{ERR} that visit the loop body at most k times. If this formula is feasible, l_{ERR} is reachable and the algorithm terminates.

For the *forward condition* (lines 12 to 14), we check in a similar manner whether the loop-head location l_{LH} is reachable at the start of the $k + 1^{st}$ loop iteration. If this is not the case, this implies that the error location is also not reachable in the $k + 1^{st}$ loop iteration (or later on), and thus the program is safe and the algorithm terminates.

For the *inductive-step case* (lines 15 to 23), we again use the CPA++ algorithm to unroll the program, though this time with a loop bound of $k + 1$ and an abstract state at the loop head as initial abstract state. For the following satisfiability check, we use the disjunction of the path formulas of all abstract states at the error location and with a loop-counter value of k (i.e., in the $k + 1^{st}$ loop iteration). Note that because we assume that the loop body cannot be reached from the error location l_{ERR}, this formula represents all paths with k safe loop iterations and a specification violation in the $k + 1^{st}$ iteration. Additionally, we strengthen the hypothesis of the inductive-step case with the currently known loop invariant that is produced by the (external) invariant generator. If the concurrently running invariant generator produces a stronger loop invariant while the inductive-step case is running, we immediately try again with the new invariant (this can be done efficiently using an incremental SMT solver). If the inductive-step case succeeds, the program is safe and the algorithm terminates. Otherwise, we repeat with a larger value of k.

Algorithm 5.3 $\mathsf{CEGAR}(\mathbb{D}, e_{INIT}, \pi_{INIT})$ for CPAs

```
Input: a composite CPA $\mathbb{D}$ that is composed of the Location CPA $\mathbb{L}$, the ARG
    CPA $\mathbb{A}$, and possibly other CPAs,
    for which $E$ denotes the set of composite abstract states and $\Pi$ the set of
    precisions,
    with additional operators fcover and refine,
    and an initial abstract state $e_{INIT} = (l_{INIT}, \cdots) \in E$
    with initial precision $\pi_{INIT} \in \Pi$
Output: false if $l_{ERR}$ is reachable, true otherwise
Variables: two sets reached and waitlist of elements of $E \times \Pi$ and
    a function $\mathsf{abort}_{ERR} : E \to \mathbb{B}$
 1: reached := $\{(e_{INIT}, \pi_{INIT})\}$
 2: waitlist := $\{(e_{INIT}, \pi_{INIT})\}$
 3: $\mathsf{abort}_{ERR} := \{(l, \cdots) \mapsto (l = l_{ERR})\}$    // $\mathsf{abort}_{ERR}$ returns true at $l_{ERR}$.
 4: loop
 5:   (reached, waitlist) := CPA++($\mathbb{D}$, reached, waitlist, $\mathsf{abort}_{ERR}$)
 6:   if $\exists((l_{ERR}, \cdots), \cdot) \in$ reached then
 7:     (reached, waitlist) := refine(reached, waitlist)
 8:     if $\exists((l_{ERR}, \cdots), \cdot) \in$ reached then
 9:       return false    // refine has detected a feasible error path.
10:   else
11:     return true
```

5.2.3. Lazy Predicate Abstraction

We can configure the Predicate CPA for classical predicate abstraction [125] with lazy abstraction [140] by making the following choices: abstraction formulas are represented with BDDs, which means that $\mathsf{stop}_{\mathbb{P}}$ is based on efficient BDD entailment checks; the $\mathsf{fcover}_{\mathbb{P}}$ operator does nothing; and $\mathsf{refine}_{\mathbb{P}}$ uses the predicate refinement strategy, which adds predicates extracted from the abstract facts to the precision. We can use any kind of abstract facts for refining the abstract model, but typically interpolants are used. In order to benefit from ABE we can use any choice for the block operator blk as long as it prevents infinitely long paths within single blocks (i.e., blk^{never} must not be used because this would lead to nontermination of the analysis). Typical choices for blk are blk^{l} and blk^{lf}, i.e., blocks end at loop-head locations and possibly on function entries and exits.

Furthermore, we wrap our CPA++ algorithm (Algorithm 5.1) inside Algorithm 5.3, which implements CEGAR by alternately calling the CPA++ algorithm in order to expand the abstract model and a refinement operator in order to refine the precision of the analysis. We give it a composite CPA that consists of the Location CPA $\mathbb{L}$, the ARG CPA $\mathbb{A}$ (necessary for constructing abstract paths during refinement), and the Predicate CPA $\mathbb{P}$. First, CEGAR uses the CPA++ algorithm in order to create the abstract model of the program. If the analysis encounters an abstract state at error location l_{ERR}, we pause the state-space exploration done by CPA++ algorithm (via the function abort_{ERR}) and start the refinement using $\mathsf{refine}_{\mathbb{P}}$. As described in Sect. 4.2, this operator reconstructs the concrete program path leading to the abstract state at l_{ERR} and checks the path for feasibility using an SMT solver. If the concrete error path is feasible, we terminate the analysis. Otherwise, the precision is refined and the CPA++ algorithm is restarted with adjusted sets reached and waitlist. This process is iterated until either a feasible concrete error path is found, or the CPA++ algorithm terminates proving the program safe.

This configuration has the effect that during refinements predicates will get added to the precision, and thus abstraction computations will occur. Both cartesian abstraction and boolean abstraction are in principle possible, however, experiments have shown that cartesian abstraction is typically too imprecise with larger block sizes (e.g., with blk^{l}) [27].

5.2.4. Lazy Abstraction with Interpolants (Impact)

We can also configure the Predicate CPA to implement lazy abstraction with interpolants [190] with the following choices: abstraction formulas are represented as SMT formulas, which means that $\mathsf{stop}_{\mathbb{P}}$ may involve SMT queries; we use interpolation as source of abstract facts during refinement; and $\mathsf{refine}_{\mathbb{P}}$ uses the Impact refinement strategy, which strengthens abstract states by conjoining the abstract facts to their abstraction formulas. To implement a behavior that is similar to the original Impact algorithm, we use the block operator blk^{SBE}, which always returns *true* and thus lets all blocks consist of only a single CFA edge (called single-block encoding). If desired, we can configure $\mathsf{fcover}_{\mathbb{P}}$ to do interpolation-based forced covering as an optimization. Just like for predicate abstraction we use Algorithm 5.3 for CEGAR with the Location CPA $\mathbb{L}$, the ARG CPA $\mathbb{A}$, and the Predicate CPA $\mathbb{P}$ as the verification algorithm.

This configuration has the effect that the precision will always stay empty and thus, the abstraction computation at block ends always trivially returns *true*.

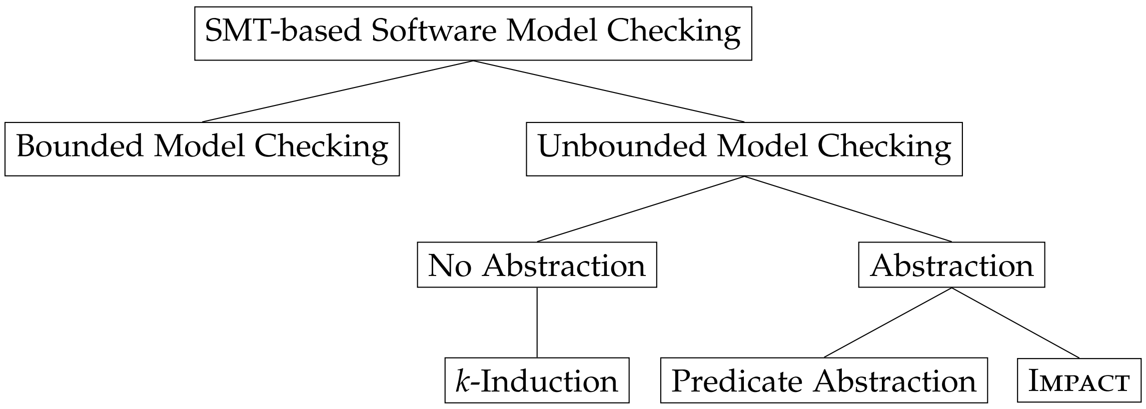


Figure 5.1.: Classification of verification approaches (taken from [28])

Instead, abstraction formulas of abstraction states will be strengthened during refinement.

Note that this configuration differs substantially from the original presentation of the Impact algorithm. To verify that we have indeed captured the essential characteristics of the approach we have also implemented the original, unchanged, Impact algorithm in the same tool as our Predicate CPA, using the same run-time environment, parser frontend, formula encoding, SMT solver, and basic optimizations like SMT query caches. Then we have compared this reimplementation of the original algorithm against our Impact-style configuration of the Predicate CPA. This comparison shows that the original Impact algorithm has similar performance characteristics as our Impact-style configuration of the Predicate CPA [52]. Thus, we are able to rely on the latter for further studying of and experimenting with lazy abstraction of interpolants.

5.3. Comparison of SMT-based Approaches for Software Verification

We are now interested in a theoretical study of the differences and similarities of the four presented approaches. We will see that having expressed the approaches in our unifying framework significantly simplifies such a study. In order to classify the four approaches, we use an existing categorization [28], which can be seen in Fig. 5.1. An experimental comparison of these four approaches also exists [28], and we will perform a further experimental study with them in Part III.

Bounded model checking is a straightforward approach for falsification, but its capability of providing safety proofs is limited. It is also the only approach that relies only on SMT solving. All the presented approaches for unbounded model checking need some other external component to be useful in practice, e.g., an invariant generator for auxiliary invariants or an interpolation engine. k-Induction differs from predicate abstraction and Impact in that it does not create an abstract model of the program. Because the safety property is often not k-inductive for the concrete program, suitable invariants need to be supplied externally for k-induction, but it is sufficient to find a k-inductive invariant, which is easier than finding a 1-inductive invariant. The CEGAR-based approaches synthesize a 1-inductive invariant themselves (the final ARG serves as such), but need to be supplied with appropriate building blocks for the invariants.

In the following, we compare the two approaches that are most similar in our unification in more detail. Predicate abstraction and the Impact algorithm are two approaches for unbounded software verification that both make use of CEGAR, SMT solving, Craig interpolation, and lazy abstraction. However, in their original presentations [35, 190] these two approaches differ substantially and it is hard to understand what the key differences and the advantages of the two approaches are. Expressing both approaches in a common framework has highlighted the essential differences and allows us to study the conceptual differences without being distracted by irrelevant differences in their presentations.

One main difference is that lazy predicate abstraction computes costly abstractions in order to have cheap coverage checks later on. This is an eager technique: computing effort is spent ahead, not knowing whether this will actually pay off. For example, along a long path within a single loop we might compute abstractions for every state, but check coverage only for the states at the loop head. On the other hand, the Impact algorithm delays all computation effort as long as possible, which means that whenever some information is needed about a state, a costly SMT-solver query needs to be made.

A further difference is how the coverage relationship is determined. In order to find as much coverage situations as possible and guarantee termination, the Impact algorithm may check coverage for a single node several times, specifically before it starts expanding nodes in the subgraph below this state after a refinement. Predicate abstraction checks coverage only once directly after the state has been created. However, states are deleted during refinement and might get rediscovered, where they are again checked for coverage.

Due to the use of an SMT solver, the coverage checks done by the Impact algorithm can be stronger than BDD-based coverage checks. Consider for example two abstract states with the abstraction formulas $x \geq 0$ and $x = 1$. An SMT solver can determine that the latter abstract state is covered by the former. If the abstraction formulas are represented by BDDs, however, each of these two abstraction formulas would be represented by a BDD that consists of a single predicate, and the BDD entailment check (which does not know the semantics of each predicate) would not be able to determine coverage.

For predicate abstraction, two common choices exist for how to compute an abstraction when creating a state: cartesian abstraction and boolean abstraction. It was shown that if using single-block encoding, boolean abstraction is too slow to be useful and only cartesian abstraction is feasible. However, the latter is imprecise if there are disjunctions in the formulas that represent program operations, because it can infer truth values only for predicates independently from each other. Disjunctions occur, e.g., if pointer-alias information is encoded in the formulas, and thus predicate abstraction with cartesian abstraction may fail to prove properties that rely on facts about pointers. Boolean abstraction can handle all boolean combinations of predicates and is thus more precise, but is only usable with large blocks. The Impact algorithm does not have this problem: it uses the interpolant directly and never loses precision. The choice of the abstraction computation of the Predicate CPA is irrelevant for Impact: both methods always abstract any formula to *true* if the set of predicates is empty.

5.4. Configurability and Extensions

The flexibility of the Predicate CPA and our unifying framework for predicate-based analyses allows us to easily create new variants of existing approaches, and to create hybrid approaches that combine core ideas from several approaches.

Note that the fact that we have expressed the core of our framework for predicate-based analyses as a CPA is already beneficial because it allows combining predicate-based approaches with other abstract domains easily [44, 47].

Such combinations of abstract domains can be more efficient than any of the abstract domains alone [8, 47]. Another advantage is that it is often possible to use the Predicate CPA as (part of) a reachability analysis in new projects and verification approaches without any need for changing the Predicate CPA itself by simply combining it with further CPAs. This has, for example, been possible for conditional model checking [36] and witness validation [29, 30].

5.4.1. Adjustable-Block Encoding for IMPACT

One example for a new combination of approaches that is enabled by our unifying framework is the use of ABE for IMPACT. By taking the configuration described in Sect. 5.2.4 for lazy abstraction with interpolants and replacing only the block operator blk^{SBE} with a different choice such as blk^{l} or blk^{lf}, we are now able (without any further theoretical or implementation effort) to use ABE and leverage the significant performance gains of it together with the IMPACT approach for which ABE was not yet available. Experiments have shown that ABE for IMPACT provides a similarly large improvement as ABE for predicate abstraction [52].

5.4.2. Flexible Bounded Analyses and Counterexample Checks

In Sect. 5.2.1 we have described how to use the Predicate CPA for bounded model checking. While bounded model checking with its loop bound k is a common instance of a bounded analysis for efficiently finding bugs, other kinds of bounded analyses may also be interesting. Our Predicate CPA in the configuration for bounded model checking is usable for any such analysis without further modification, as long as it is combined with another CPA that limits the analyzed state space in the desired way instead of the Loop-Bound CPA. Just like the latter, such state-space-restricting CPAs need to have an unsound transfer relation that ensures that no infinitely long paths will be unrolled by the CPA algorithm. Possible choices for bounds beyond loop iterations include bounds on the length of paths, number of calls to certain functions (useful for bounded verification of callers of a certain API), etc.

Another scenario where a specific bounded analysis is useful is checking error paths that were found by other (imprecise) analyses [53]. If it is known that a found error path might be infeasible and is finite, it is possible to combine the

Predicate CPA in its configuration for bounded model checking with a CPA that restricts the analyzed state-space to exactly the potential error path. This is used, for example, in the tool CPAchecker as a so-called counterexample check to increase the precision of certain otherwise imprecise configurations, such as those based on tracking explicit values of variables (which cannot efficiently track facts about nondeterministic variables and large heap structures) [100, 186, 231]. This is similar to the validation of violation witnesses [30], but for witnesses we cannot assume that the analyzed state space is finite and thus we need to resort to a configuration for unbounded model checking.

5.4.3. Further Configuration Options

Note that most of the configuration options of the Predicate CPA are orthogonal to the choices made for implementing one of these two approaches. In particular, this is true for the more low-level "technical" options like the choices for the encoding of program semantics discussed in Sect. 4.4 or the used SMT solver, which do not affect the course of the algorithms. Implementing such options once immediately makes them usable with any approach expressed within our unifying framework and thus opens up the possibility for the study of new combinations. We will use this for our experimental evaluation in Part III.

5.4.4. Sequential Combinations of Configurations

From a theoretical point of view, the Predicate CPA in a configuration for unbounded software verification with the most precise encoding of program semantics is more powerful than other configurations such as those for bounded model checking, or those with an approximation of program semantics. Unfortunately, however, in practice it is not always possible to use such a configuration with precise semantics, for two reasons. The first reason is performance: Solving formulas over bitvectors, floating-point numbers, and arrays may be a more difficult task for an SMT solver than solving for example a similar formula (of course with different semantics) that has only linear arithmetic and no arrays. The increased solving time for more precise formulas might in practice make it infeasible to verify larger and more complex programs. The second reason are specific problems with interpolation. For example, the SMT theory of arrays does not guarantee the existence of quantifier-free interpolants [161]. Even if (quantifier-free) interpolants exist, not all SMT solvers support interpolation for more complex theories such as the theory of floating-point numbers, or fail for some queries (cf. Sects. 6.3.1 and 16.1). The choice of which SMT solver is used may also depend on reasons such as availability on a specific platform, license, or further supported features, and thus we should not assume to always have an SMT solver that supports all necessary theories. Of course, *k*-induction does not suffer from problems with interpolation, but *k*-induction-based approaches need an additional invariant generator [31], which might also fail to provide the necessary invariants. However, note that in the configuration of the Predicate CPA for counterexample checks, without abstraction computations, refinements, and interpolation, such that the SMT solver has to solve only a single query about the feasibility of a finite set of program paths, these issues may not apply at all or their performance cost may be acceptable.

Thus, in practice a combination of configurations can be beneficial and easily implemented on top of our unifying framework. We can run an imprecise configuration such as one based on linear arithmetic first, verify its result using a precise bounded configuration, and if the result was wrong, start a full analysis with the precise configuration. Depending on the performance difference between the configurations and the cost of verifying the first result, this can pay off. Furthermore, this will allow us to avoid potential problems with interpolation in the precise configuration as far as possible, because for verifying whether an existing result is valid it is not needed.

Such a sequential combination of predicate-based configurations (with counterexample checks as described in Sect. 5.4.2 but no checks of safety proofs, and in combination with other non-predicate-based configurations as well) was for example successful in several years of the International Competition on Software Verification [100, 186, 231], showing that this strategy can indeed be beneficial in practice. Figure 5.2 illustrates the sequential combination of five verification approaches that was used in SV-COMP'15 [100]: besides two configurations of an explicit-value analysis [44], our predicate analysis was used in its configurations for *k*-induction and predicate abstraction, both with an approximation of program semantics using only linear arithmetic. The last configuration of the sequential combination was again the configuration for predicate abstraction, but this time with bitprecise semantics. If any of these analyses found a safety proof

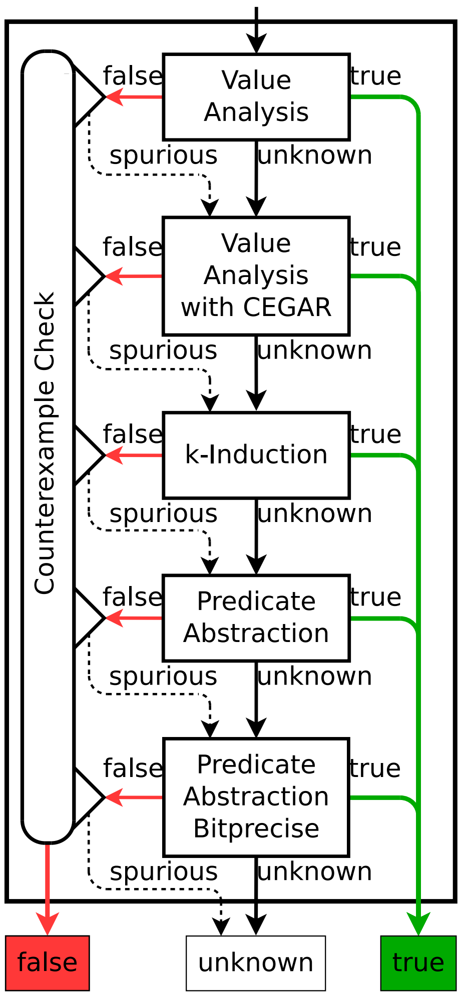


Figure 5.2.: Sequential combination of algorithms as used in SV-COMP'15 submission (adapted from [100])

or a specification violation that was confirmed by a counterexample check, the whole analysis terminated immediately. If, however, an analysis did not find a result within some time limit, or found a specification violation that could not be confirmed (e.g., because of the imprecise approximation using linear arithmetic), the next analysis of the sequence was used.

6. Implementation in CPAchecker

In this chapter, we describe the state of the implementation of our framework. The complete implementation is part of the software-verification framework CPAchecker [41] and available in the trunk of its repository [1]. CPAchecker is licensed under the permissive Apache 2.0 license.

We have extended an existing implementation of a CPA for predicate abstraction with ABE [42, 187] to support all the features and components from Fig. 4.1, with only a few exceptions that we discuss below. We also provide an implementation of the CPA++ algorithm (Algorithm 5.1) and of the necessary algorithms for k-induction (Algorithm 5.2) and CEGAR (Algorithm 5.3), such that all configurations from Chapter 5 are usable. As BDD library we use JavaBDD [2] and for SMT solving several solvers are available, which we discuss in Sect. 6.3. All choices of components are configurable via individual options, such that they can be combined flexibly. Fig. 6.1 provides an overview of the architecture of CPAchecker with these components for predicate analysis (components for other kinds of analyses are available but not shown). Furthermore, there are algorithms for counterexample checks (cf. Sect. 5.4.2), and sequential combinations of analyses (cf. Sect. 5.4.4) available, which can optionally be enabled.

6.1. Discovery Strategies for Abstract Facts

As explained in Sect. 4.2.3, there exist several strategies for the discovery of abstract facts during refinement. Interpolation, which is the most common approach, is fully supported in our implementation by delegating to an SMT solver and is the default strategy. Path invariants have been implemented and any kind of analysis that is available as a CPA in CPAchecker can be employed

[1] `https://cpachecker.sosy-lab.org/download.php`
[2] `http://javabdd.sourceforge.net/`

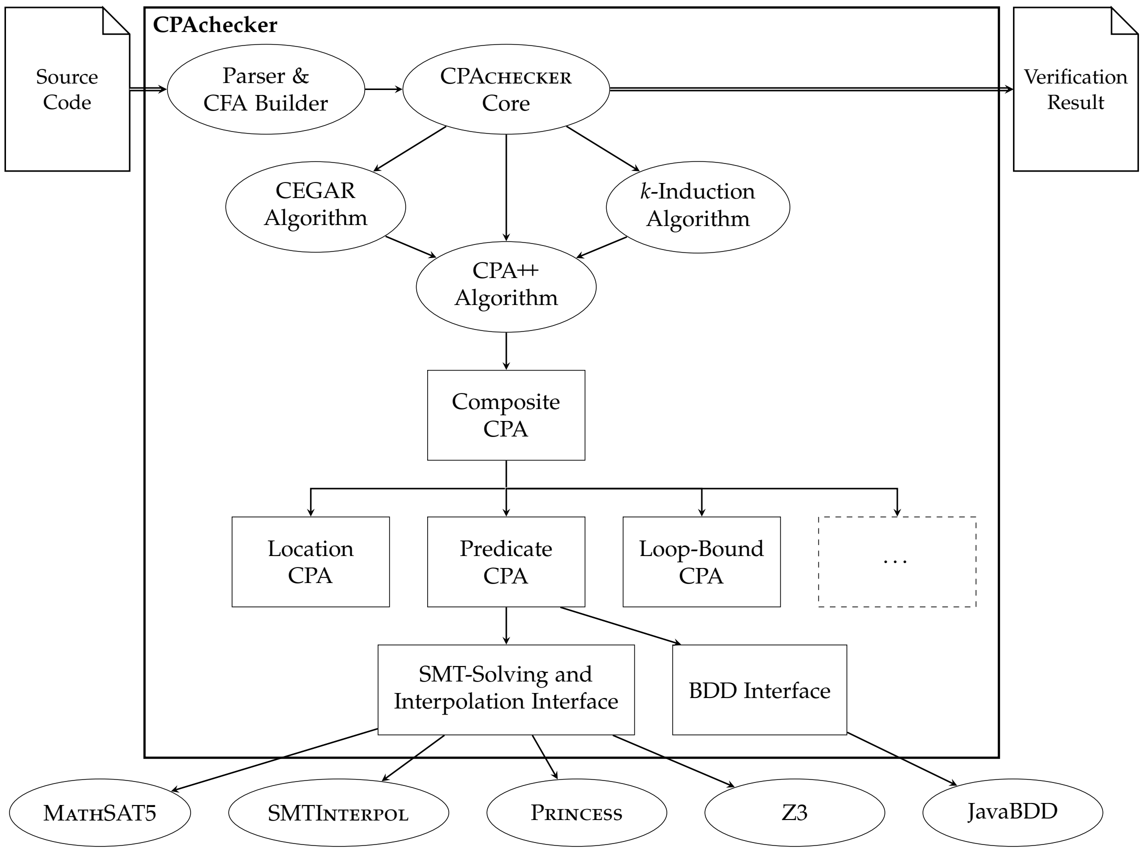


Figure 6.1.: Architecture of CPAchecker with components for predicate analysis (adapted from [41])

as an invariant generator [220]. In case the invariant generator does not succeed to find invariants that are strong enough to refute the current spurious abstract counterexample within its resource limit, the abstract-fact discovery falls back to interpolation. There exists a heuristic for generating predicates statically from the program, however, it produces only the condition of the last assume statement on the error path as predicate, and thus is of limited use. This heuristic can only be used for predicate refinement in combination with interpolation or path invariants. Predicate-discovery strategies that are based on weakest preconditions are not implemented.

6.2. Strongest-Postcondition Operator

As described in Sect. 4.4, various possible choices exist for representing semantics of C programs in formulas. Our implementation supports all theory combinations from Table 4.2 on page 51. Thus, we can choose whether we want to support an unbounded heap and bitprecise semantics of arithmetic operators. This implementation includes contributions from Mikhail Mandrykin (pointer handling with uninterpreted functions [186]) and from the Bachelor's theses of Matthias Dittrich (bitprecise handling of integers [108]) and Stephan Lukasczyk (pointer handling with arrays or quantifiers [188]). Furthermore, a more fine-grained memory model based on disjoint regions is also available [7, 228].

For the approximation of integer types with linear arithmetic (cf. Sect. 4.4.1) the sound encoding with additional disjunctions can be used, or a direct approximation of bitvector operators with their linear counterparts. The latter is unsound but produces formulas that are hopefully easier to solve. The approximation of floats with linear arithmetic is always unsound in our implementation.

The encoding of the heap (cf. Sect. 4.4.3) simulates a separate contiguous memory region for each different type of the C programming language. Each cell of such a memory region stores a single value of the respective type, i.e., the heap is not organized as an array of bytes.

6.3. SMT Solvers

We have integrated a range of SMT solvers. One benefit of being able to choose between solvers is that different solvers have different performance characteristics, and thus having several of them available allows finding and choosing an efficient one, not only based on general performance figures such as provided by the International Satisfiability Modulo Theories Competition (SMT-COMP)[3] [90], but specifically for our workload. Furthermore, the supported theories and features of SMT solvers differ and some SMT solvers work better in practice for certain operations than others (for example, with fewer failures), so it can be necessary to choose a certain SMT solver depending on the analysis configuration. Unfortunately, there is no "one-size-fits-all" solution.

[3] `http://www.smtcomp.org`

Table 6.1.: Interpolating SMT solvers

	MathSAT5[a] [81]	Princess[b] [211]	SMTInterpol[c] [77]	Z3[d] [101]
License	proprietary[e]	LGPL v3	LGPL v3	MIT
Institution	Fondazione Bruno Kessler, University of Trento	Uppsala University	University of Freiburg	Microsoft Research
Language	C++	Scala	Java	C++
Quantifier-free theories supported[f]	A, BV, FP, LIA, LRA, UF	A, LIA, UF	A, LIA, LRA, UF	A, BV, FP, LIA, LRA, UF, others
Quantifier support	✗	✓	✗	✓

[a] `http://mathsat.fbk.eu`
[b] `http://www.philipp.ruemmer.org/princess.shtml`
[c] `https://ultimate.informatik.uni-freiburg.de/smtinterpol`
[d] `https://github.com/Z3Prover/z3`
[e] research and evaluation purposes only, exceptions on request
[f] A: Arrays; BV: Bitvectors; FP: Floating-Point; LIA: Linear integer arithmetic; LRA: Linear real arithmetic; UF: Equality over uninterpreted functions

Additionally, for interpolation queries the SMT solvers do not differ only in performance and feature support, but also in their answers, because for a given query in general an infinite amount of possible interpolants exist. Thus, we are interested in studying interpolants from different solvers and learning which kind of interpolants may be well-suited for verification.

We are interested in SMT solvers that support interpolation and are available as a library (i.e., are not coupled to a specific verification framework). The license should allow at least academic experimentation with the solver. There are a few solvers that satisfy these criteria, however, not all of them are actually usable in our use case. The solver OpenSMT2 [146] supports the theories of uninterpreted functions and linear real arithmetic, but only individually and not in combination, and thus would not be able to solve queries about arithmetic with heap values. The theorem prover Vampire [171] also supports some features of an SMT solver including interpolation [144] and even participates in SMT-COMP. However, its support for interpolation over theories is limited and it cannot generate models for theories like integers, which means we would not be able to report a concrete error path if a specification violation is found. The solver OptiMathSAT [216] is an extension of MathSAT5 for optimization modulo theories, but does not differ from MathSAT5 in the features we are interested in, so we use only MathSAT5 itself. There are four remaining solvers, which we list in Table 6.1. All these solvers support the SMT-LIB standard [16] and are integrated in CPAchecker such that they are usable with the Predicate CPA.

6.3.1. Comparison

The solvers from Table 6.1 differ in several ways that can influence the choice between them. For MathSAT5, a specific license needs to be requested for all purposes except research and evaluations, whereas all other solvers are available under a permissive license and can be used in all circumstances.

The fact that MathSAT5 and Z3 run as native code outside the control of the JVM complicates the configuration of memory limits. For optimal performance of tools running on the Java virtual machine (JVM), users need to specify the maximum size of the JVM heap. Typically we would set the maximum heap size to a value just below the amount of available memory. However, because of the greedy JVM memory allocation strategies, this would lead to total memory usage beyond what is available (and thus a crash), if a native library with significant memory consumption is used. Therefore, users need to estimate the amount of memory the solver will need and prevent the JVM from using too much memory by setting a smaller heap size upfront (the maximum heap size cannot be changed dynamically). An inappropriate partitioning of the memory between solver and JVM can lead to crashes or decreased performance due to unnecessary GC work.

MathSAT5 and Z3 support (among others) the theories of bitvectors and floats, whereas Princess and SMTInterpol only support linear arithmetic. Coincidentally, this means that currently it is necessary to use a native solver for bitprecise verification. Princess and Z3 support quantifiers. In MathSAT5, support for interpolation over the theory of floats is missing, and support for interpolation over a combination of bitvectors and uninterpreted functions is limited.[4] In Z3,

[4] Source: communication with Alberto Griggio, cf. also our evaluation in Sect. 16.1.

a number of bugs in the interpolation engine exist that can lead for example to crashes. We have reported these bugs in July 2016, but until June 2017 they have not been fixed.[5] In general, it seems that interpolation support for Z3 is currently not maintained.[6] Apart from the mentioned problems, all four solvers are under active development and the issues we reported typically got fixed in a timely manner.

6.3.2. Integration in CPAchecker

The integration of all four SMT solvers in CPAchecker was designed to be as efficient as possible, in order to exploit the full potential of each SMT solver. Thus, we integrated them as libraries instead of starting the solver's binary as a separate process, with which communication would be based on strings and be more costly. For the solvers written in non-JVM-based languages (MathSAT5, Z3), we use Java bindings based on the Java Native Interface (JNI), which is the fastest way to call native code from Java code (all other methods use JNI under the hood because this is the only way that is directly supported by the JVM, so they cannot be faster).

For all integrated solvers, there is only a small wrapper layer that implements a common interface and forwards all method calls directly to the solver via its API, in order to not unfairly discriminate against some solvers. This means for example that we never construct and store formula ASTs in CPAchecker itself. Instead we use methods offered by the solver to construct formulas inside the solver's memory, and we store only references to the solver's internal formulas. This avoids needing to copy formulas (repeatedly) between CPAchecker and the solver via creating and parsing strings. Instead, formulas are created only once, which for example allows the solver to keep internal state related to the formulas if they are reused in consecutive queries. Whenever a necessary feature is supported directly by an SMT solver, we made sure to use the solver's internal implementation, otherwise we reimplement it on top of the solver's API. This is done for example for AllSMT [177] (getting all satisfying assignments of an

[5] list on `https://github.com/Z3Prover/z3/issues?q=interpolation%20created:2016-06..2016-07`

[6] No relevant commits have been made to the directory `src/interp` of the Z3 repository since 2015.

SMT formula with respect to a set of predicates), which is necessary for boolean predicate abstraction.

Recently, this common interface and wrapper layer for SMT solvers was extracted from CPAchecker into a separate project called JavaSMT[7] [163] and can now be used in other tools as well. During this process, the solver integration was improved by the maintainer of JavaSMT, George Karpenkov, especially with regards to the integration of Z3.

[7] `https://github.com/sosy-lab/java-smt`

7. Related Work

Unfortunately, while general overviews over approaches for model checking exist [153], there exists little work on unification of such approaches on a theoretical level. The CPA concept [39], on which our framework for predicate analysis is based, is one such unification of algorithms for model checking and for data-flow analysis. For hardware model checking, several variants and extensions of the IC3 algorithm were unified in a common framework and studied [129]. In the following we focus on individual approaches for software model checking that are related to the approaches that we have expressed in our unifying framework and their extensions, and on verification tools that combine several algorithms on the implementation level.

7.1. SMT-Based Algorithms for Software Model Checking

There are other approaches for software verification besides the four that we unify in this work. In the following, we briefly discuss the most important SMT-based approaches, ordered roughly accordingly to how similar they are to the approaches that we have discussed so far.

The Ufo algorithm [2] combines the Impact algorithm with predicate abstraction. Ufo is similar to Impact, but implements a choice between performing predicate-abstraction computation when creating fresh abstract states, and initializing them with *true* as Impact does. Refinement is done using interpolation, and the interpolants can be used to either strengthen the abstract states (pure Impact behavior), or to update the set of predicates (pure predicate-abstraction behavior), or do both. This approach can be seen as an instantiation of our verification framework with a refinement operator that uses interpolants as abstract facts and both the Impact- and the predicate-refinement strategies (cf. Sect. 4.2.4).

Symbolic execution [168] follows each path in the program separately and interprets its operations; the abstract states track explicit and symbolic values of program variables in a symbolic store as well as constraints over the symbolic values. If a variable is assigned a nondeterministic value, a fresh symbolic value is stored; if an explicit value can be determined by the analysis, then the explicit value is stored. Constraints that are encountered along a path are tracked and checked for satisfiability, using the symbolic store as interpretation, whenever the feasibility of the path needs to be determined (e.g., if an error location is reached). The framework presented in this work can be configured as an analysis that behaves similarly to symbolic execution (just without symbolic store) by using the CPA algorithm with the Predicate CPA configured to use blk^{never} and merge^{sep} instead of $\mathsf{merge}_{\mathbb{P}}$. The operator blk^{never} has the effect of disabling abstraction computations and thus accumulating the semantics of all program operations of a path in the path formula of abstract states during traversal (as for BMC). The operator merge^{sep} has the effect of preventing all merges between abstract states and thus keeping all paths separate, forming a reachability tree. Note that differently from symbolic execution this configuration tracks all values syntactically.

Slicing abstractions [68, 114] (a.k.a. "state splitting") starts with an abstract reachability graph (ARG) in which all abstract states are labeled with *true*. The algorithm iteratively searches for an infeasible error path in this graph and computes interpolants for the respective path. The strategy for refining the abstract model consists of duplicating each abstract state for which an interpolant was found (including its edges), and conjoining the interpolant to one of the resulting abstract states and the negated interpolant to the other one ("state splitting"). Then all edges of both resulting states are checked for feasibility. This always results in enough edges being removed such that the current infeasible error path no longer exists in the ARG. This is repeated (CEGAR) until either no infeasible error path exists anymore, or a feasible error path is found. The approach of splitting abstract states has also been extended to a combination of predicate abstraction and explicit-value analysis [133], similar to the combination of lazy predicate abstraction and explicit-value analysis [44].

Trace abstraction [137] is a CEGAR-based approach in which the iteratively refined abstract model of the program is not a set of abstract states, but instead an automaton that represents an overapproximation of the feasible paths of the program. Every time a spurious counterexample is detected, a trace automaton that represents a set of infeasible paths including the current counterexample is created using interpolation, and this trace automaton is subtracted from the current abstract model.

Software proof-based abstraction with counterexample-based refinement (SPACER) [169] is an approach that combines CEGAR with its dual, proof-based abstraction (PBA) [192]. While CEGAR maintains an overapproximation of the program and refines it using infeasible error paths, PBA maintains an underapproximation and refines it if it finds a safety proof that holds only for the underapproximation but not for the original system. SPACER follows the PBA approach but uses an abstraction of the underapproximation to allow handling infinite-state systems, and refines this abstraction using CEGAR.

Model checking modulo theories (MCMT) [122, 123] is an approach that focuses on verifying infinite-state systems that use arrays. It is based on a backwards-reachability analysis and SMT solving for theories that fulfill certain conditions. MCMT has been combined with CEGAR and interpolation to define an analysis that can be described as a backwards variant of Impact, and applied to software model checking [5]. This approach uses interpolation to compute quantifier-free interpolants for a restricted class of formulas with arrays, and can prove universally quantified properties over arrays automatically.

IC3 [1] [62], which is also known as property-directed reachability (PDR) [113], is an algorithm for model checking finite-state systems. It aims at producing an inductive invariant that is strong enough to prove safety by incrementally learning clauses that are inductive with regard to the previously learned clauses. Such clauses are derived by generalizing from counterexamples to induction proofs. PDR was originally designed for boolean transition systems and based on SAT solving. It has been generalized from boolean systems to SMT [143], and applied to software in various ways [55, 78, 80, 156], which we discuss in the following. If PDR is combined with an explicit (instead of symbolic) tracking of the program counter, this lets the algorithm produce an abstract reachability tree [78]. In fact, because the sets of clauses that PDR learns fulfill the properties of interpolants, this tree-based PDR can even be seen as a version of Impact, just with a different way of producing interpolants. A hybrid approach that uses both

[1] Short for "Incremental Construction of Inductive Clauses for Indubitable Correctness"

a regular interpolation engine as well as PDR for producing interpolants is also possible [78]. It would be an interesting extension of our Predicate CPA to adopt the clause-learning strategy of PDR as a discovery strategy for abstract facts during refinement (cf. Sect. 4.2.3). Another approach for software verification using PDR is to define a boolean abstract model of the program using predicate abstraction, and use an almost unchanged PDR algorithm for verifying the abstract model [80]. The abstraction is refined using typical predicate-discovery strategies (e.g., interpolation) whenever an infeasible error path is found. CTIGAR [55] is an approach for applying PDR to software that does not rely on CEGAR (i.e., using error paths for refinement), but uses counterexamples to induction (CTI) for abstraction refinement. CTIGAR computes abstract CTIs from the concrete CTIs of PDR by using predicate abstraction, and refines the abstraction using interpolation if it finds a clause that is inductive with regard to the previously learned clauses, but its abstract version is not. PDR can also be extended from standard induction to property-directed *k*-induction [156]. This allows it to more easily verify programs for which useful 1-inductive invariants are cumbersome and difficult to find, while more concise *k*-inductive invariants exist.

Loop invariants that are strong enough to verify program safety can also be computed via abduction [107]. Similar to the PDR-based approaches, a candidate invariant is strengthened until it becomes inductive. However, while PDR starts from facts that are known to hold, the abductive approach starts from the conjecture it wants to prove and asks an abduction engine to generate candidate strengthenings that would allow the conjecture to hold. Then it needs to check whether one of the candidates holds, which may need further recursive strengthenings with backtracking. As abduction engine, it is possible to use for example quantifier elimination in Presburger arithmetic.

7.2. Extensions of the Studied Approaches

In addition to the verification algorithms of the previous section, there exists also a number of techniques that extend or improve some part of an existing algorithm without proposing a completely new verification approach. Often such extensions can actually be applied more generally than for the algorithm they have been originally presented for. One of the advantages of a flexible framework such as the Predicate CPA is indeed that orthogonal extensions need to be implemented only once and can immediately be applied and studied with a wide range of approaches. In the following, we discuss a few of the most important extensions that are or could be integrated in our framework.

Large-block encoding [27] groups loop-free blocks of program operations together in order to reduce the number of solver queries and refinements. Our Predicate CPA supports its generalization adjustable-block encoding [42] and relies on it for encoding a bounded unrolling of the whole program in a single block for bounded model checking.

In the framework presented here, we use invariants computed by an abstract interpreter for strengthening k-induction-based proofs (cf. Sect. 5.2.2). It is also possible to use invariants for strengthening the transition relation of other approaches, e.g., for predicate abstraction [149]. The difference is that, in the former case, the invariants are crucial for the power of the verification approach, i.e., they allow verification of programs that cannot be verified with k-induction without invariants [31, 110], whereas in the latter case, the invariants serve only to increase the efficiency of the approach by reducing the number of necessary predicates. Such a use of invariants could be combined with the Predicate CPA as well, and it would be interesting to see how this compares to using an invariant generator as source of abstract facts during refinement (cf. Sect. 4.2.3).

Several proposals have been made to extend the approach of lazy abstraction with interpolants to recursive programs, including nested interpolants [138] and the two similar algorithms Whale [3] and Duality [193]. While we do not address the verification of recursive programs in this work, it is possible to use block-abstraction memoization [233] to extend any CPA-based analysis to support recursion [118], and use nested interpolants for extending the refinement operator of the Predicate CPA [100]. Verification of recursive programs can also be reduced to iterative verification of a series of nonrecursive programs, and this has been successfully used together with the Predicate CPA in its configuration of lazy predicate abstraction [75, 76].

Besides the verification of recursive programs, many approaches have been proposed in the last years for improving the effectiveness and efficiency of counterexample-based refinement and interpolation strategies. Such techniques are often orthogonal to the actual verification algorithm as long as it is based on CEGAR, and could be combined with or implemented in our Predicate CPA.

Path slicing [152] attempts to simplify interpolation queries by omitting irrelevant facts of the infeasible error path.

Which interpolants are used for refining the abstract model during the analysis can be crucial, and it can be beneficial to generate several interpolants for a given error path and choose an appropriate one. This can be done inside the interpolation engine by combining proof transformations and labeling functions [112], on top of the interpolation engine by exploring a lattice of abstracted interpolants [182], or as an extension of the refinement operator with refinement selection [47]. For the latter case, we extract multiple infeasible paths from the error path (e.g., using sliced-path prefixes [48]), refine each of them in the regular way, and then select an appropriate refinement using some heuristic. Refinement selection works independently of how possible refinements are found, i.e., it also works with refinements that are not based on interpolation and across different abstract domains.

Other techniques that attempt to improve the usefulness of interpolants for the purpose of verification are for example counterexample minimization [5] and the computation of “simple” interpolants [214]. However, in general, it is not known how to compute the best or at least good interpolants for a given error path, and the existing techniques are heuristics that may not always provide an optimal solution. Our presented framework for software verification provides a platform for research in this direction by allowing to easily add new such approaches and study their behavior together with several different verification algorithms.

7.3. Software Verifiers

The number of tools for automatic software verification might be even larger than the number of verification algorithms. A good overview of tools for verification of C programs is given by the International Competition on Software Verification [24]. However, software verifiers that are suited only for specific use cases, focus on some particular verification approach, or are prototypical implementations, are not suited for our goal of studying and comparing algorithms. Thus, we discuss here only the more mature verification frameworks that implement several verification approaches (in alphabetic order).

The CProver framework[2] is the base of a number of bitprecise verification tools, for example 2LS[3] [64], which implements BMC and *k*-induction, as well as abstract interpretation with several abstract domains, Cbmc[4] [86], which implements BMC, Satabs[5] [87], which implements predicate abstraction, and Wolverine[6] [175], which implements the Impact algorithm. However, even while there is an underlying common framework, each approach is implemented in a different tool and without unification of approaches. In contrast, our unifying framework and the implementation in a single tool allow to combine and mix different approaches more flexibly.

Another tool that is based on the CProver framework is Esbmc[7] [96], which implements BMC and *k*-induction [195]. In contrast to the aforementioned CProver-based tools, Esbmc is backed by an SMT solver and can use Boolector, CVC4, MathSAT5, Yices, and Z3. Esbmc supports bitprecise verification as well as approximation with linear arithmetic using the SMT theories QF_AUFBV and QF_AUFLIRA. The extension DepthK[8] [208] of Esbmc strengthens its *k*-induction procedure with invariants generated upfront by static analysers [207].

F-Soft [148] is a proprietary model checker that implements SAT-based bounded model checking as well as predicate abstraction with refinement via weakest preconditions. It also generates invariants using abstract interpretation with domains such as intervals or octagons for strengthening the analysis [149].

SeaHorn[9] [132] is a verification framework that transforms the program that should be verified into a set of Horn clauses. These Horn clauses are then verified by a backend, for which multiple choices exist, including the PDR implementation in Z3 [143] and an implementation of the SPACER algorithm [169]. SeaHorn can also use the abstract-interpretation framework IKOS [65], either as verification backend or for computing invariants that are supplied to any of the other backends.

[2] http://www.cprover.org
[3] http://www.cprover.org/2LS
[4] http://www.cprover.org/cbmc
[5] http://www.cprover.org/satabs
[6] http://www.cprover.org/wolverine
[7] http://www.esbmc.org
[8] https://github.com/hbgit/depthk
[9] https://seahorn.github.io

The Smack[10] [73, 205] toolchain combines a frontend that translates C programs into the Boogie intermediate verification language [103] with existing verifiers for Boogie. Such possible backends for Smack include Corral [179], which implements a variant of BMC, and Duality [193], which implements an interprocedural extension of Impact.

The software-verification framework Ufo[11] [4] implements the Ufo algorithm described in Sect. 7.1, which combines Impact and predicate abstraction, and supports recursive programs using the Whale algorithm [3]. Additionally, it provides abstract interpretation using the domains of intervals and intervals with disjunctions (boxes) as an alternative to predicate abstraction [131], and uses interpolation to refine these domains [1].

The program-analysis framework Ultimate[12] consists of a number of plugins that can be combined into full toolchains for tasks like software verification. Specific configurations of Ultimate include Ultimate Automizer [136], which implements trace abstraction [137], Ultimate Kojak [202], which is based on an algorithm called Impulse that is similar to splitting abstract states [114], and Ultimate Taipan [126], which combines trace abstraction and abstract interpretation. All these approaches make use of CEGAR and nested interpolation [138], which extends interpolation to trees of formulas to allow verification of recursive programs.

[10] `https://github.com/smackers/smack`
[11] `https://bitbucket.org/arieg/ufo`
[12] `https://ultimate.informatik.uni-freiburg.de`

Part II.

Reliable Benchmarking

Contents

8. Motivation 93

8.1. Overview 95

8.2. Restrictions 96

9. Requirements for Reliable Benchmarking 99

9.1. Measure and Limit Resources Accurately 100

9.1.1. Measuring CPU Time and Wall Time 100

9.1.2. Measuring Peak Memory Consumption 100

9.2. Terminate Processes Reliably 101

9.3. Assign Cores Deliberately 101

9.4. Respect Nonuniform Memory Access 103

9.5. Avoid Swapping 103

9.6. Isolate Individual Runs 104

10. Limitations of Existing Methods 105

10.1. Measuring Resources May Fail 105

10.1.1. Measuring CPU Time and Wall Time 105

10.1.2. Measuring Peak Memory Consumption 106

10.2. Enforcing Limits May Fail 106

10.3. Termination of Processes May Fail 107

10.4. Hardware Allocation May be Ineffective 108

10.5. Isolation of Runs may be Incomplete 109

11. Impact of Hardware Characteristics on Parallel Tool Executions 111

11.1. Overview of Hardware Characteristics 111

11.2. Experiment Setup 112

11.3. Impact of Hyperthreading 113

11.4. Impact of Shared Memory Bandwidth and Caches 114

11.5. Impact of Turbo Boost 115
11.6. Impact of NUMA 116
11.7. Impact of Multiple CPUs 117
11.8. Investigation of Impact of Multiple CPUs 119

12. State-of-the-Art Benchmarking with Cgroups and Containers 123
12.1. Introducing Cgroups for Benchmarking 123
12.2. Benchmarking Containers Based on Namespaces 126

13. BenchExec: A Framework for Reliable Benchmarking 131
13.1. System Requirements 131
13.2. Benchmarking a Single Run 132
13.3. Benchmarking a Set of Runs 134
13.4. Comparison with Requirements for Reliable Benchmarking 137
13.5. Discussion 139
13.6. Encouraging Replicable Experiments 141

14. Related Work 143
14.1. Benchmarking Strategies 143
14.2. Benchmarking Tools 144

8. Motivation

Performance evaluation is an effective and inexpensive method for assessing research results [224], and in some communities, like high-performance computing[1], transactional processing in databases[2], natural-language requirements processing[3], and others, performance benchmarking is standardized. Performance benchmarking, i.e., measuring execution time, memory consumption, and other performance characteristics of the tool for a large set of input files is also a standard practice for the evaluation of tools and approaches for automatic software verification. Benchmarking is used by researchers for comparing different tools, evaluating and comparing different features or configurations of the same tool, or for finding out how a single tool performs on different inputs. Also verification competitions, like the International Competition on Software Verification (SV-COMP) [24], require exact measuring of resource consumption for hundreds or thousands of runs for each participating tool. For this thesis, we want to use benchmarking to evaluate our unifying framework for predicate-based software verification from Part I and compare our implementation against other tools.

To recover from its replication crisis [91, 172, 206], experimental computer science needs a stronger focus on replicability [67, 157, 227]. While replicability requires several properties to be satisfied (e.g., documentation of experimental setup, availability of data, repeatability of experiments), in this part we focus on the technical aspects of benchmarking that can render the results invalid. To be able to perform replicable performance experiments, we need a benchmarking infrastructure that guarantees that the results are obtained by reliable and valid measurements.

An experiment is *replicable* if it is guaranteed that a different research team is able to obtain the same results later again by rerunning the benchmarks on

[1] `https://www.spec.org`
[2] `http://www.tpc.org`
[3] `http://nlrp.ipd.kit.edu`

a machine with the same hardware and the same software versions (cf. ACM's guideline [4]). Replication of experimental results requires reliable *measurement*. We call a measurement *reliable*, if the measurement method ensures *accuracy* (small systematic and random measurement error, i.e., no bias or "volatile" effects, resp.) and sufficient *precision* [150] (cf. also ISO 3534-2:2006). While measuring execution time may appear trivial, a closer look reveals that quite the contrary is the case. In many circumstances, measuring the wall time, i.e., the elapsed time between start and end of a tool execution, is insufficient because this does not allow a meaningful comparison of the resource usage of multithreaded tools, and may be inadvertently influenced by input/output operations (I/O). Measuring the CPU time is more meaningful but also more difficult, especially if child processes are involved. Furthermore, characteristics of the machine architecture such as hyperthreading or nonuniform memory access can *nondeterministically* affect results and need to be considered carefully in order to obtain accurate results. Obtaining reliable measurement values on memory consumption is even harder, because the memory that is used by a process may increase or decrease at any point in time.

Besides measuring, also the ability to *limit* resource usage (e.g., memory consumption) of a tool during benchmarking is a requirement for replicable experiments. Here it is necessary to ensure for example that limits on memory consumption are not exceeded *at any point in time* during the execution of the tool. Child processes again add further complications.

For replicable experiments it is also important that the tool executions are properly *isolated*. Executing the benchmarked tool should leave the system in an unchanged state, and parallel tool executions must not affect each other due to, for example, mutual interference of processes from different tool executions or contention with regard to shared hardware resources. Another important aspect is the potentially huge heterogeneity between different tools in a comparison: tools are written in different programming languages, require different libraries, may spawn child processes, write to storage media, or perform other I/O. All of this has to be considered in the design of a benchmarking environment, ideally in a way that does not exclude any relevant tool from being benchmarked.

[4] `https://www.acm.org/publications/policies/artifact-review-badging`

Unfortunately, existing benchmarking tools do not always ensure that these issues are handled and that the results are reliable. Thus, we have found that in order to being able to properly perform our planned experimental evaluation, with our own tool as well as other tools, a benchmarking infrastructure is necessary that allows us to reliably measure and limit resources and makes it as easy as possible to obtain and present scientifically valid experimental data.

8.1. Overview

We present the following solutions and insights towards reliable benchmarking for all scenarios that are described above:

- We define a set of necessary requirements that need to be fulfilled for reliable benchmarking (Chapter 9).
- We show that some existing methods for resource measurements and limitations do not fulfill these requirements and lead to invalid experimental results in practice (Chapter 10).
- We investigate the impact of benchmarking multiple tool executions in parallel and report experimental results on measurement errors, depending on certain hardware characteristics (Chapter 11).
- We describe how to implement a benchmarking environment on a Linux system which fulfills all mentioned requirements (Chapter 12).
- We introduce the open-source implementation BenchExec, a set of ready-to-use tools that fulfill the requirements for reliable benchmarking. These tools are also used successfully in practice by other research groups and competitions [23] (Chapter 13).

A publication that contains this part of this thesis was accepted with minor revisions for publication [49]. A preliminary version (without the isolation of runs described in Sect. 12.2, and the discussion of hardware influences in Chapter 11) was already published before [46].

8.2. Restrictions

In order to guarantee reliable benchmarking, we need to introduce a few restrictions. We only consider the benchmarking of tools that adhere to the following restrictions: the tool (1) is CPU-bound, i.e., if compared to CPU usage, input and output operations from and to storage media are negligible, and input and output bandwidth does not need to be limited nor measured (this assumes the tool does not make heavy use of temporary files); (2) executes computations only on the CPU, i.e., does not make use of separate coprocessors such as GPUs, (3) does not require external network communication during the execution; (4) does not spread across several machines during execution, but is limited to a single machine; and (5) does not require user interaction.

These restrictions are well-justified and acceptable for our use case of benchmarking tools for automatic software verification. Reading from storage media (1), apart from the input file, is not expected for verification tools. In case a tool produces much output (e.g., by creating large log files), this would primarily have a negative impact on the performance of the tool itself, and thus does not need to be restricted by the benchmarking environment. Sometimes, I/O cannot be avoided for communicating between several processes, however, for performance this should be done without any actual storage I/O anyway (e.g., using pipes). Note that RAM disks should not be used as temporary storage on benchmarking systems, because their usage would neither be counted in memory measurements nor be restricted by the memory limit. If a tool executes computations on coprocessors like GPUs (2), this kind of resource consumption would also need to be measured in order to get meaningful results. This is out of our scope, and also more complex than measurements on CPUs, because GPUs and similar coprocessors are architecturally much more diverse then CPUs and no common hardware-independent interface exists for them. Not supporting external network communication (3) is necessary, because it is possible for a tool to offload work to remote servers [34, 226], and this would mean to exclude the offloaded work from benchmarking. Benchmarking a distributed tool (4) is much more complex and out of scope. However, techniques and ideas from this work as well as our benchmarking framework can be used on each individual host as part of a distributed benchmarking environment. User interaction (5) is generally not supported for benchmarking.

While we consider a proper isolation of the executed tool in order to prevent accidental sabotage of the measurements or other running processes, we do not focus on security concerns, i.e., we assume the executed tool does not maliciously try to interfere with the benchmarking. We also do not consider the task of providing the necessary execution environment, i.e., the user has to ensure that the tool itself and all necessary packages and libraries that are required to execute the tool are available in the correct versions. Furthermore, we assume that enough memory is installed for the operating system (OS), the benchmarking environment, and the benchmarked process(es) without swapping, and that no CPU-intensive tasks are running outside the control of the benchmarking environment. All I/O is assumed to be local, because network shares can have unpredictable performance.

9. Requirements for Reliable Benchmarking

There exist three major difficulties that we need to consider for benchmarking. The first problem is that a tool may arbitrarily spawn child processes, and a benchmarking framework needs to control this. Using child processes is common practice. For example, verifiers might start preprocessors, such as `cpp`, or solvers, like an SMT-backend, as child processes. Some tools start several child processes, each with a different analysis or strategy, running in parallel, while some verifiers spawn a separate child process to analyze counterexamples. In general, a significant amount of the resource usage can happen in one or many child processes that run sequentially or in parallel. Even if we know that our own tool does not start child processes, for comparing with other tools we still need to use a generic benchmarking framework that handles child processes correctly.

The second problem occurs if the benchmarking framework assigns specific hardware resources to tool runs, especially if such runs are executed in parallel and the resources need to be divided between them. Machine architectures can be complex and a suboptimal resource allocation can negatively affect the performance and lead to nondeterministic and thus nonreplicable results. Examples for differing machine architectures can be seen in Appendix A and online[1].

The third problem arises with ensuring the independence of different tool executions. In most cases, benchmarks consist of a large number of tool executions, each of which is considered to be independent from the others. For accurate results, each tool execution should be performed in isolation, as on a dedicated machine without other tool executions, neither in parallel nor in sequential combination, for example to avoid letting the order of tool executions influence the results.

[1] `https://www.sosy-lab.org/research/benchmarking`

1. Measure and Limit Resources Accurately
2. Terminate Processes Reliably
3. Assign Cores Deliberately
4. Respect Nonuniform Memory Access
5. Avoid Swapping
6. Isolate Individual Runs

Figure 9.1.: Requirements for reliable benchmarking

We have identified six requirements (shown in Fig. 9.1) that address these problems and need to be followed for reliable benchmarking. This list can also serve as a checklist not only for researchers who use benchmarking, but also for assessing the quality of experimental results in research reports. In the following, we explain each requirement in more detail.

9.1. Measure and Limit Resources Accurately

9.1.1. Measuring CPU Time and Wall Time

The CPU time of a tool must be measured and limited accurately, including the CPU time of all (transitive) child processes that the tool started. The wall time (i.e., the elapsed time between start and end of the tool execution) must be measured without being affected by changes to the system clock, e.g., due to daylight-savings time or due to time adjustments in the background that are for example caused by NTP services.

9.1.2. Measuring Peak Memory Consumption

For benchmarking, we are interested in the peak resource consumption of a process, i.e., the smallest amount of resources with which the tool could successfully be executed with the same result. Thus, the memory usage of a process is defined as the peak size of all memory pages that occupy some system resources. This means, for example, that we should not measure and limit the size of the address space of a process, because it may be much larger than the actual memory usage. This might happen due to memory-mapped files or due to allocated but unused memory pages (which do not actually take up resources, because the Linux kernel lazily allocates physical memory for a process only when a virtual memory page is first written to, not when it is allocated). The size of the heap can also not be used as memory measure because it does not include the stack, and the so-called resident set of a process (the memory that is currently kept in RAM) cannot be used because it does not include pages that are in use but have been swapped out.

If a tool spawns several processes, these can use shared memory, such that the total memory usage of a group of processes is less than the sum of their individual memory usages. Shared memory occupies system resources only once and thus needs to be counted only once by the benchmarking framework.

Explicitly setting a limit for memory usage is important and should always be done, because otherwise the amount of memory available to the tool is the amount of free memory in the system, which varies over time and depends on lots of external factors, preventing repeatable results.

9.2. Terminate Processes Reliably

If a resource limit is violated, it is necessary to reliably terminate the tool including all of its child processes. Even if the tool terminates itself, the benchmarking environment needs to ensure that all child processes are also terminated. Otherwise a child process could keep running and occupy CPU and memory resources, which might influence later benchmarks on the same machine.

9.3. Assign Cores Deliberately

Special care is necessary for the selection of CPU cores that are assigned to one tool execution. For the scheduler of the OS, a core is a processing unit that allows execution of one thread independently of what happens on a different core. However, on the hardware level cores are usually not fully independent because of shared hardware resources. In this case, the performance of a core and thus the CPU-time and wall-time measurements of a tool execution are influenced by the actions of threads running on other cores, and the performance impact depends not only on the characteristics of the machine's hardware, but also on the type of operations performed by all these threads, and on the timing of their operations relative to each other. For example, if all threads are heavily accessing the memory at the same time, a larger influence is to be expected than if the threads happen to access the memory at different times. Because we cannot guarantee an upper bound on the size of the performance influence and because it happens nondeterministically, we need to avoid such performance influences as far as possible in order to achieve accurate measurements.

If the benchmarked tool is concurrent and should be executed using several cores, avoiding performance influences between the tool's *own* threads and processes is specific to the tool and thus the responsibility of the tool itself. For performance influences on the tool from *other* processes, the most reliable way to avoid them would be to execute only one instance of the benchmarked tool at the same time, and ensure that no other processes outside the control of the benchmarking environment are active. However, this might not always be possible, for example due to machines being shared with other users, or due to the amount of benchmarking work being so large that parallel executions are required. In such cases, the benchmarking environment should assign a fixed set of CPU cores to each (parallel) tool execution and not let the scheduler of the OS assign cores dynamically, in order to prevent additional performance influences from processes being moved around. The benchmarking environment needs to compute the mapping of CPU cores per run such that the CPU cores for one run are as close as possible and the sets of cores for separate runs are as independent as possible on the hardware level[2], and thus the performance impact is minimized. Users need to know about the characteristics of their benchmarking machine and what kind of performance influences they need to expect, in order to properly decide how many parallel executions are acceptable for a given benchmark.

For example, we would usually consider it acceptable to have parallel tool executions on different CPUs that each have their own local memory, whereas it would usually considered to be *not* acceptable to have parallel tool executions on different virtual cores of the same physical core (just like there should never be two simultaneous tool executions sharing one virtual core). More information on

[2] I.e., with high cohesion and loose coupling.

how hardware characteristics can affect performance is given in Chapter 11. In any case, the used resource allocation should be documented and made available together with the benchmark results.

9.4. Respect Nonuniform Memory Access

Systems with several CPUs often have an architecture with nonuniform memory access (NUMA), which also needs to be considered by a benchmarking environment. In a NUMA architecture, a single CPU or a group of CPUs can access parts of the system memory locally, i.e., directly, while other parts of the system memory are remote, i.e., they can only be accessed indirectly via another CPU, which is slower. The effect is that once a process has to access remote memory, this leads to a performance degradation depending on the load of the inter-CPU connection and the other CPU. Hence, a single tool execution should be bound to memory that is local to its assigned CPU cores, in order to avoid nondeterministic delays due to remote memory access.

9.5. Avoid Swapping

Swapping out memory must be avoided during benchmarking, because it may degrade performance in a nondeterministic way. This is especially true for the benchmarked process(es), but even swapping of an unrelated process can negatively affect the benchmarking, if the benchmarked process has to wait for more free memory to become available. Absolutely preventing swapping can typically only be done by the system administrator by turning off all available swap space. In theory, it is not even enough to ensure that the OS, the benchmarking environment, and the benchmarked processes all fit into the available memory, because the OS can decide to start swapping even if there is still memory available, for example, if it decides to use some memory as cache for physical disks. However, for benchmarking CPU-bound tools, with high CPU and memory usage, and next to no I/O, this is unlikely to happen with modern OS. Thus, the main duty of the benchmarking environment is to ensure that there is no overbooking of memory, and that memory limits are enforced effectively. It is also helpful if the benchmarking environment monitors swap usage during benchmarking and warns the user of any swapping.

9.6. Isolate Individual Runs

If several tool executions are executed in parallel, and to some extent even if they are executed sequentially, the different instances of the benchmarked tool(s) can interfere with each other, which could influence the performance and/or change the results.

One common reason for mutual interference are write accesses to shared files in the temp directory and in the home directory. For example, if a tool uses a temporary file with a fixed name, a cache directory, or configuration files in the home directory, parallel instances may interfere with each other nondeterministically. Even if runs are executed strictly sequentially, left-over files from previous runs could influence later runs if the tool reads these files, and prevent repeatability of experiments (because results then depend on the order of executing the runs).

Another reason for mutual interference of parallel runs are signals like `SIGKILL` or `SIGTERM`, if they get sent to processes that belong to a different tool instance. This may happen inadvertently, for example in a well-meaning cleanup script that tries to terminate child processes of a tool with the command `killall`.

The benchmarking environment should isolate the benchmarked processes to prevent such interference.

10. Limitations of Existing Methods

Some of the methods that are available on Linux systems for measuring resource consumption and for enforcing resource limits of processes have several problems that make them unsuitable for benchmarking, especially if child processes are involved. Any benchmarking environment needs to be aware of these limitations and avoid using naive methods for resource measurements. However, later on in Sect. 14.2 we will see that indeed a number of existing benchmarking tools use the methods described in this section and thus do not ensure reliable results.

10.1. Measuring Resources May Fail

10.1.1. Measuring CPU Time and Wall Time

Measuring wall time is sometimes done by reading the system clock at start and end of a run and calculating the difference. However, because the system clock can jump due to time adjustments and even change its pace, it is important to use a time source that is guaranteed to be strictly monotonic and of constant rate. Many operating systems and programming languages offer such a time source with high precision specifically for benchmarking.

Measuring CPU time of the main process of a tool, for example using the tool `time` or a variant of the system call `wait` (which returns the CPU time after the given process terminated), does not reliably include the CPU time of child processes that were spawned by the main process. The Linux kernel only adds the CPU time used by child processes to that of the parent process *after* the child process has terminated *and* the parent process waited for the child's termination with a variant of the system call `wait`. If the child process has not yet terminated or the parent did not explicitly wait for its termination, the CPU time of the child is lost. This is a typical situation that might happen for example if a verifier starts an SMT solver as a child process and communicates with the solver via `stdin` and `stdout`. When the analysis finishes, the verifier would terminate the solver process, but usually would not bother to wait for its termination. A tool that runs different analyses in parallel child processes would also typically terminate as soon as the first analysis returns a valid result, without waiting for the other analyses' termination.[1] In these cases, a large share of the total CPU time is spent by child processes but not included in the measurement.

10.1.2. Measuring Peak Memory Consumption

Some measurement tools only provide a view on the current memory usage of individual processes, but we need to measure the *peak* usage of a *group* of processes. Calculating the peak usage by periodically sampling the memory usage and reporting the maximum is inaccurate, because it might miss peaks of memory usage. If the benchmarked process started child processes, one has to recursively iterate over all child processes and calculate the total memory usage. This contains several race conditions that can also lead to invalid measurements, for example, if a child process terminates before its memory usage could be read. In situations where several processes share memory pages (e.g., because each of them loaded the same library, or because they communicate via shared memory), we cannot simply sum up the memory usage of all processes. Thus, without keeping track of every memory page of each process, manually filtering out pages that do not occupy resources because of lazy allocation, and counting each remaining page exactly once, the calculated value for memory usage is invalid. If all this is done with a high sampling frequency (to not miss short peaks of memory usage), we risk that the benchmarked process is being slowed down by the increased CPU usage.

10.2. Enforcing Limits May Fail

For setting resource limits, some users apply the tool `ulimit`, which uses the system call `setrlimit`. A limit can be specified for CPU time as well as for memory, and the limited process is forcefully terminated by the kernel if one

[1] The organizers of SV-COMP'13 experienced this for a portfolio-based verifier. Initial CPU-time measurements were significantly too low, which was only discovered by chance. The verifier had to be patched to wait for its subprocesses and the benchmarks had to be rerun.

of these limits is violated. However, similar to measuring time with the system call `wait`, limits imposed with this method affect only individual processes, i.e., a tool that starts n child processes could use n times more memory and CPU time than allowed. Limiting memory is especially problematic because either the size of the address space or the size of the data segment (the heap) can be limited, which do not necessarily correspond to the actual memory usage of the process, as described in Sect. 9.1.2. Limiting the resident-set size (RSS) is no longer supported.[2] Furthermore, if such a limit is violated, the kernel terminates only the one violating process, which might not be the main process of the tool. In this case it depends on the tool itself how such a situation is handled: it might terminate itself, or crash, or even continuously respawn the terminated child process and continue. Thus, this method is not reliable.

It is possible to use a self-implemented limit enforcement with a process that samples CPU time and memory usage of a tool with all its child processes, terminating all processes if a limit is exceeded, but this is inaccurate and prone to the same race conditions, as described above for memory measurement.

10.3. Termination of Processes May Fail

In order to terminate a tool and all its child processes, one could try to (transitively) enumerate all its child processes and terminate each of them. However, finding and terminating all child processes of a process may not work reliably for two reasons. First, a process might start child processes faster than the benchmarking environment is able to terminate them. While this is known as a malicious technique ("fork bomb"), it may also happen accidentally, for example due to a flawed logic for restarting crashed child processes of a tool. The benchmarking environment should guard against this, otherwise the machine might become unusable. Second, it is possible to "detach" child processes such that they are no longer recognizable as child processes of the process that started them. This is commonly used for starting long-running daemons that should not retain any connection to the user that started them, but also might happen incidentally if a parent process is terminated before the child process. In this case, an incomplete benchmarking framework could miss to terminate child processes.

[2] `http://man7.org/linux/man-pages/man2/setrlimit.2.html`

The *process groups* of the POSIX standard (established with the system call `setpgid`[3]) are not reliable for tracking child processes. A process is free to change its process group, and tools using child processes often use this feature.

10.4. Hardware Allocation May be Ineffective

There are two mistakes that can be made when attempting to assign a specific set of CPU cores to a benchmark run. First, the set of CPU cores for a process can be specified with the tool `taskset` or alternatively by the system call `sched_setaffinity`. However, a process can change this setting freely for itself, and does not need to follow predefined core restrictions. Thus, this is not a reliable way to enforce a restriction of the CPU cores of a process.

Second, the numbering system of the Linux kernel for CPU cores is complex and not consistent across machines of different architectures (not even with the same kernel version). Any "naive" algorithm for assigning cores to tool executions (e.g., allocating cores with consecutive ids), will fail to produce a meaningful core assignment, and give suboptimal performance, on at least some machines.

The Linux kernel assigns numeric ids for CPUs (named *physical [package] id*), for physical cores (named *core id*), and for virtual cores (named *processor id* or *CPU id*). The latter is used for assigning cores to processes. Additionally there is a numeric id for NUMA memory regions (named *node id*), which is used for restricting the allowed memory regions of a process. There is no consistent scheme how these ids are assigned by the kernel. In particular, we have found the following intuitive assumptions to be *invalid*:

- CPUs, virtual cores, and NUMA regions are always numbered consistently (instead the virtual cores or NUMA regions may be numbered in a different order than the CPUs).
- The core id is assigned consecutively (instead there may be gaps).
- Virtual cores that belong to the same physical core always have processor ids that are as far apart as possible.

[3] `http://man7.org/linux/man-pages/man2/setpgrp.2.html`

- Virtual cores that belong to the same physical core always have processor ids that are as close as possible (i.e., consecutive).
- Virtual cores that belong to the same physical core have the same core id.
- Virtual cores that belong to different physical cores have different core ids (instead, on systems with several NUMA regions per CPU, there are several physical cores on each CPU with the same core id).
- The tuple (physical [package] id, core id) uniquely identifies physical cores of the system.

For several of these invalid assumptions, a violation can be seen in Fig. A.1 or Fig. A.2 in Appendix A.

Therefore we must not rely on any assumption about the numbering system and only use the explicit CPU topology information given by the kernel. The authoritative source for this information is the `/sys/devices/system/cpu/` directory tree. Note that the file `/proc/cpuinfo` neither contains the node id for NUMA regions nor the information which virtual cores share the same hardware, and thus cannot be used to compute a meaningful core assignment for benchmarks.

10.5. Isolation of Runs may be Incomplete

Processes can be isolated from external signals by executing each parallel tool execution under a separate user account. This also helps to avoid influences from existing files and caches in the user's home directory, but it does not allow separating the temporary directory for each run, and thus parallel runs can still influence each other if they use temporary files with hard-coded names. Many tools allow using the environment variable `TMPDIR` for specifying a directory that is used instead of `/tmp`, but not all tools support this option. For example, the Java VM (both Oracle and OpenJDK) ignores this variable.

11. Impact of Hardware Characteristics on Parallel Tool Executions

Because we often have a large set of independent tool executions to perform (e.g., several tool configurations on many input files), we can save a significant amount of time if we can execute the tool several times in parallel. However, certain hardware characteristics of today's machines can influence the performance of tools that are executed in parallel on one machine. Thus, it is important to understand the sources of undesired performance influences and how to minimize their impact, in order to decide whether parallel executions are acceptable. In this chapter, we present an overview of important hardware characteristics and highlight the effects they can have on parallel tool executions based on experimental results.

11.1. Overview of Hardware Characteristics

In today's machines, CPU cores can typically be roughly organized in a hierarchy with three layers:[1] A machine may have multiple *CPUs* (also named sockets or [physical] packages), each CPU may have multiple *physical cores*, and each physical core may have multiple *virtual cores* (also named [hardware] threads or processors). The virtual cores of one physical core may share some execution units, level-1, and level-2 caches (e.g., in case of hyperthreading). The physical cores of one CPU may share a level-3 cache and the connection to the RAM. The CPUs of one machine may share the connection to the RAM and to I/O. If not all cores on the system share the same connection to the RAM, the system is said

[1] Systems can be even more complex and have more layers, however, the hierarchy presented here captures the facts that are most important for the performance of software from our target domain. Thus, we use this abstracted definition and nomenclature.

to have a NUMA architecture. Appendix A shows examples of two hardware architectures.

The "closer" two cores are, i.e., the more hardware they share, the larger can be the mutual performance influence of two threads running on them in parallel. The largest influence is to be expected by hyperthreading (several virtual cores sharing execution units) or the "module" concept of AMD CPUs (several virtual cores sharing level 1 and 2 caches). Even if we use separate physical cores for each tool execution, there can be further nondeterministic influences on performance. Modern CPUs often adjust their frequency by several hundred MHz depending on how many of their cores are currently used, with the CPU running faster when less cores are used (this is commonly called "Turbo Boost" or "Turbo Core"). For memory-intensive programs, the influence by other processes running on different physical cores that share the same level 3 cache and the connection to the RAM (and thus compete for memory bandwidth) can also be significant.

On the other hand, the closer two cores are, the faster they can typically communicate with each other. Thus, we can reduce performance impact from communication by allocating cores that are close together to each tool execution.

11.2. Experiment Setup

To show that these characteristics of the benchmarking machine can significantly influence the performance, and thus have a negative influence on benchmarking if not handled appropriately, we conducted several experiments on a machine with two Intel Xeon E5-2650 v2 CPUs and 68 GB of RAM per CPU. The CPUs (each with eight physical cores) support hyperthreading (with two virtual cores per physical core) and Turbo Boost (base frequency is 2.6 GHz, with up to 3.4 GHz if only one core is used). The machine has a NUMA architecture. A graphical overview of this system can be seen in Fig. A.2 of Appendix A.

As an example for a benchmarked tool we took the verifier CPAchecker[2] in revision 17 829 from the project repository[3] with its configuration for predicate analysis. The machine was running a 64-bit Ubuntu 14.04 with Linux 3.13 and OpenJDK 1.7 as Java virtual machine. As benchmark set we used 4 011 C pro-

[2] `https://cpachecker.sosy-lab.org`
[3] `https://svn.sosy-lab.org/software/cpachecker/trunk`

grams from SV-COMP'15 [22] (excluding categories not supported by this configuration of CPAchecker). In order to measure only effects resulting from concurrency between parallel tool executions and avoid effects from concurrency inside each tool execution, we restricted each tool execution to one virtual core of the machine. We also limited each tool execution to 4.0 GB of memory and 900 s of CPU time. Except were noted, Turbo Boost was disabled such that all cores were running at 2.6 GHz. Apart from our tool executions, the machine was completely unused. All experiments were conducted with BenchExec 1.2. We show the accumulated CPU time for a subset of 2 414 programs that could be solved by CPAchecker within the time and memory limit on the given machine. (Including timeouts in the accumulated values would skew the results.) Time results were rounded to two significant digits. Tables with the full results and the raw data are available online.[4]

Note that the actual performance impact of certain hardware features will differ according to the characteristics of the benchmarked tool and the benchmarking machine. For example, a tool that uses only little memory but fully utilizes its CPU core(s) will be influenced more by hyperthreading than by nonlocal memory, whereas it might be the other way around for a tool that relies more on memory accesses. In particular, the results that are shown here for CPAchecker and our machine are not generalizable and show only that there *is* such an impact. Because the quantitative amount of the impact is not predictable and might be nondeterministic, it is important to avoid these problems in order to guarantee reliable benchmarking.

11.3. Impact of Hyperthreading

To show the impact of hyperthreading, which is also named simultaneous multithreading, we executed the verifier twice in parallel on one CPU of our machine. In one part of the experiment, we assigned each of the two parallel tool executions to one virtual core from separate physical cores of the same CPU. In a second part of the experiment, we assigned each of the two parallel tool executions to one virtual core from the same physical core, such that both runs had to share the hardware resources of one physical core. A scatter plot with the results is shown

[4] `https://www.sosy-lab.org/research/benchmarking/#benchmarks`

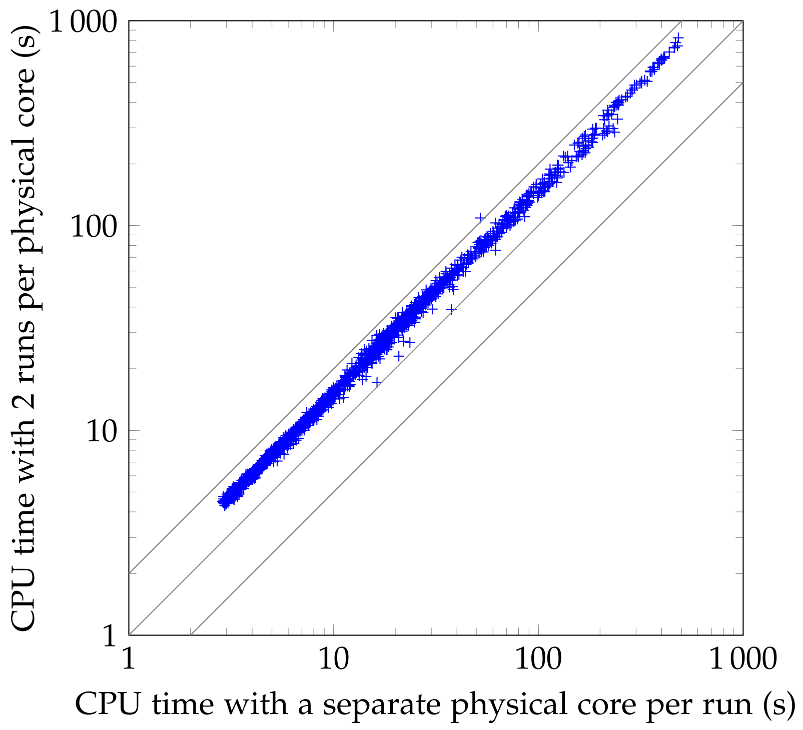


Figure 11.1.: Scatter plot showing the influence of hyperthreading for 2 414 runs of CPAchecker: the data points above the diagonal show a performance decrease due to an inappropriate assignment of CPU cores

in Fig. 11.1. For the 2 414 solved programs from the benchmark set, 16 hours of CPU time were necessary using two separate physical cores and 25 hours of CPU time were necessary using the same physical core, an increase of 53 % caused by the inappropriate core assignment. This shows that hyperthreading can have a significant negative impact, and parallel tool executions should not be scheduled on the same physical core.

11.4. Impact of Shared Memory Bandwidth and Caches

To show the impact of a shared memory connection and a shared level-3 cache for multiple physical cores, we experimented with 1 to 8 parallel tool executions on the same CPU (each on its own physical core), i.e., with 1 to 8 used physical cores that share a common level-3 cache and the memory bandwidth. The second virtual core of every physical core was unused, and Turbo Boost was disabled.

A plot with the results is shown in Fig. 11.2 (shaded blue data points). For the 2 414 solved programs from the benchmark set, 16 hours of CPU time were necessary if only one physical core was used, whereas 20 hours of CPU time were necessary with eight physical cores used; the increase of 22 % is caused by the contention on cache and memory accesses. The linear-regression line in the plot shows that the used CPU time scales linearly with the amount of used cores. This can be explained by the CPU dynamically allocating equal shares of its memory bandwidth and level-3 cache to each of those cores that are actively used.

This experiment shows that parallel tool executions on the same CPU can influence the measured CPU time significantly. However, because the wall time that is necessary for benchmarking can be decreased drastically by using parallel tool executions, it might in practice sometimes be necessary to compromise and use at least a few parallel tool executions during benchmarking. Note that by utilizing only a subset of the cores of a CPU the size of the undesired influence on CPU time can be reduced while keeping most of the wall-time savings compared to no parallelization at all.

11.5. Impact of Turbo Boost

To show the impact of Turbo Boost, we executed benchmarks with the same setup as in Sect. 11.4, but now with Turbo Boost enabled. This means that the CPU uses a higher frequency depending on the current load of its cores. Without Turbo Boost, a used core of this CPU always runs at 2.6 GHz. With Turbo Boost, a single used core of this CPU can run at 3.4 GHz, if the CPU is otherwise idle, and even if all eight cores are used, they can still run at 3.0 GHz. The results are also shown in Fig. 11.2 (red data points). As expected, due to the higher frequency, the CPU time is lower than with Turbo Boost disabled, and the more physical cores are used in parallel, the higher the used CPU time becomes. The latter effect is larger if Turbo Boost is enabled than if Turbo Boost is disabled, because in addition to the contention of cache and memory bandwidth, there is now the additional performance influence of the varying CPU frequency. Instead of increasing by 22 %, the used CPU time now increases by 39 % (from 13 to 18 hours) if using eight instead of one physical core. Thus, a dynamic scaling of the CPU frequency should be disabled if multiple tool executions run in parallel on a CPU.

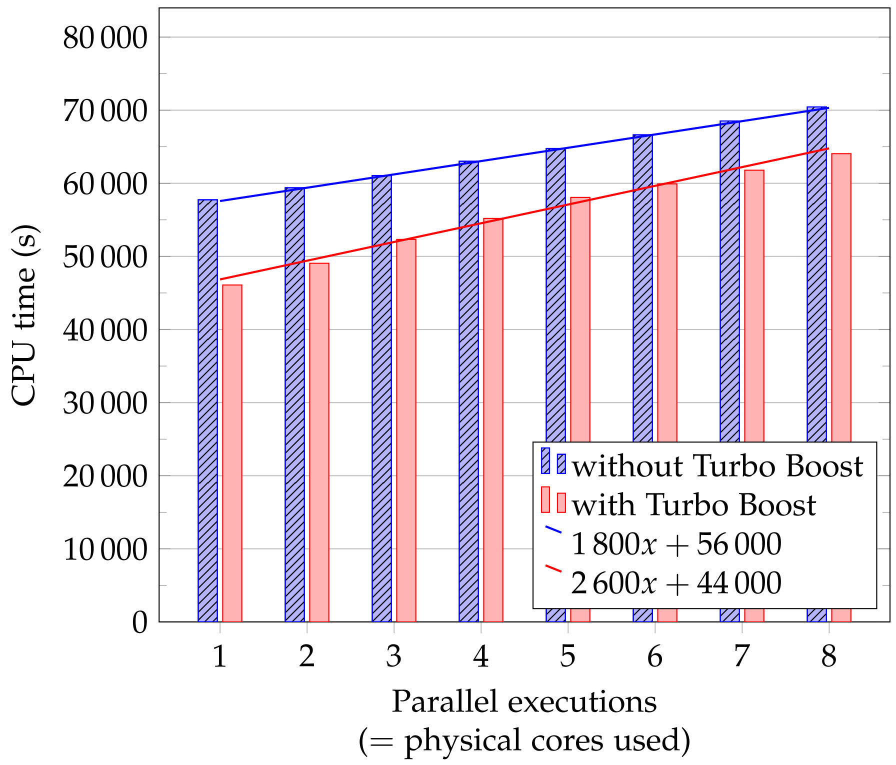


Figure 11.2.: Plot showing the influence of Turbo Boost, shared Level 3 cache, and shared memory bandwidth for 2 414 runs of CPACHECKER

11.6. Impact of NUMA

To show the impact of NUMA, we executed 16 instances of the verifier in parallel, one instance per physical core of the two CPUs of our machine. In one part of the experiment, we assigned memory to each tool execution that was local to the CPU that the tool was executed on. In a second part of the experiment, we deliberately forced each of the 16 tool executions to use only memory from the *other* CPU, such that all memory accesses were indirect. For the 2 414 solved programs from the benchmark set, 23 hours of CPU time were necessary using local memory and 24 hours of CPU time were necessary using remote memory, an increase of 6.8 % caused by the inappropriate memory assignment. While the performance impact in this case is not that large, there exists no reason to not ensure a proper memory assignment in the benchmarking environment and rule out this influence completely.

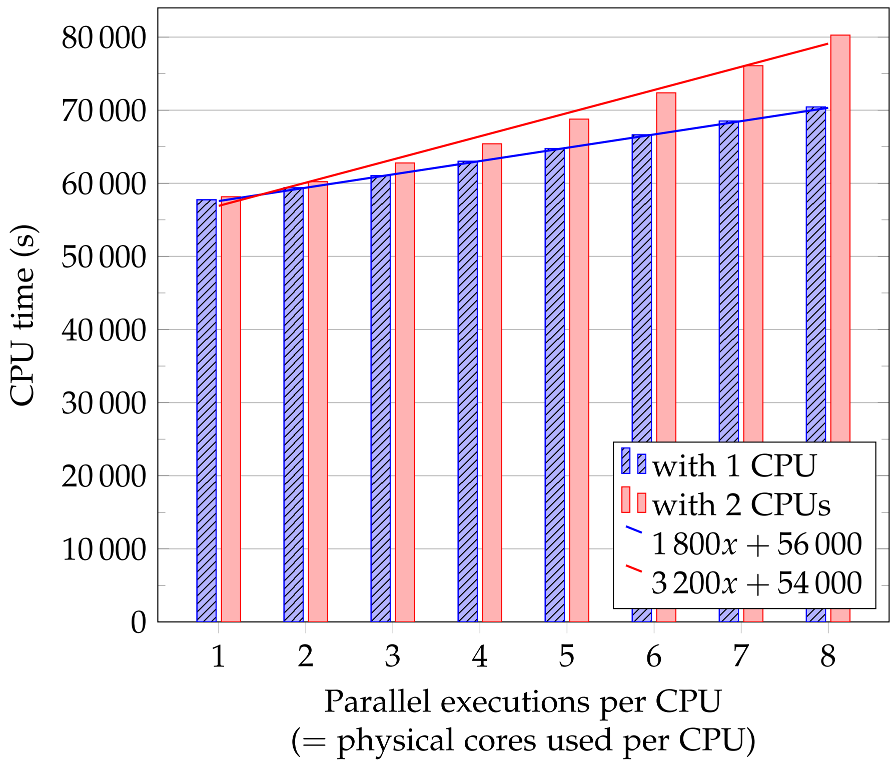


Figure 11.3.: Plot showing the influence of using multiple CPUs for 2 414 runs of CPAchecker

11.7. Impact of Multiple CPUs

To show the impact of multiple CPUs used in parallel, we experimented with 1 to 16 parallel tool executions across all CPUs. For one part of the experiment, we used the same setup as in Sect. 11.4: using only one CPU and executing 1 to 8 parallel tool executions, each on a designated physical core (hence, the shaded blue bars in Fig. 11.3 match exactly the shaded blue bars in Fig. 11.2). For a second part of the experiment, we used both CPUs of the machine, executing 1 to 8 parallel tool executions *on each CPU*, i.e., with 2 to 16 parallel tool executions on the machine. A summary of the results is shown in Fig. 11.3. If only one physical core of the whole machine is used, 16 hours of CPU time were necessary for the 2 414 tool executions. This increases linearly to 20 hours of CPU time if all eight cores of one CPU are used (shaded blue data points in the plot). If one physical core of each of the two CPUs is used (i.e., two parallel tool executions),

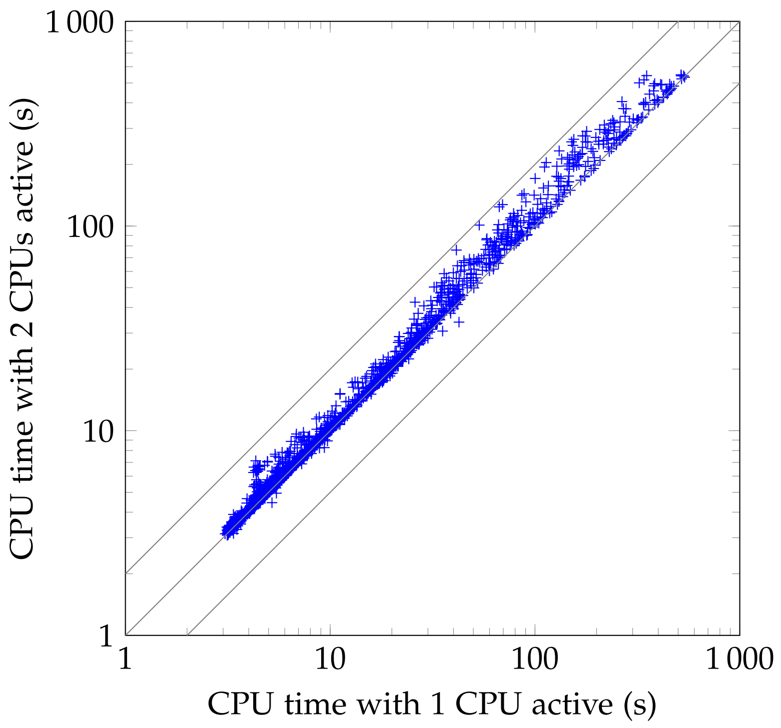


Figure 11.4.: Scatter plot showing the influence of using multiple CPUs for 2 414 runs of CPAchecker: the data points above the diagonal show a performance decrease due to using two CPUs in parallel instead of only one CPU

the necessary CPU time is also 16 hours: there is no significant difference to using one physical core of only one CPU. However, if more physical cores per CPU are used, using n physical cores of each of the two CPUs with $2n$ parallel tool executions (red data points) is slower than using n physical cores of only one of the CPUs. The maximal difference occurs for eight cores per CPU, which uses 22 hours of CPU time compared to 20 hours (an increase of 14 %).

We also show a scatter plot in Fig. 11.4 with more detail on the last part of the experiment, i.e., for 8 and 16 physical cores per CPU, respectively. It shows that for some of the 2 414 runs, the performance was actually equal in both cases (the data points on the diagonal), whereas for others the CPU time is almost doubled (the data points close to the gray line for $y = 2x$).

This experiment shows that parallelization of tool executions across multiple CPUs at least sometimes needs to be treated similarly to parallelization of tool executions across multiple cores of the same CPU.

11.8. Investigation of Impact of Multiple CPUs

Because we did not expect the performance impact of using multiple CPUs with separate local memory for independent parallel tool executions, we tried to find out the source of this effect using additional experiments and to ensure that other possible performance influences were ruled out as far as possible. Unfortunately we did not find an explanation so far, and the respective machine is no longer available for further experiments, so the reason for the performance impact of multiple CPUs remains unknown to us. However, we document these experiments here in the hope that this will help others to investigate this issue in the future.

We repeated the complete experiment from Sect. 11.7 at least two times, and the performance variation across the three experiments was negligible compared to the difference between using one or two CPUs. We also tried to reproduce this effect on two other multi-CPU machines, both with AMD CPUs (four Opteron 6380 and eight Opteron 8356, respectively). In both cases there was no significant performance change for varying the number of used CPUs. Unfortunately, we did not have access to other multi-CPU machines with Intel CPUs. So we were not able to reproduce the effect on the AMD CPUs.

Back to the machines with Intel CPUs, the effect of decreased performance if both CPUs are active did not occur in experiments with a tool that uses mainly the CPU and not much memory. To measure this, we experimented with the same setup for the tool `bc` (an arbitrary-precision calculator) and let it compute π with 1 000 to 19 890 digits using the formula $\arctan(1) \times 4$. This takes between 0.43 s and 880 s on our machine and uses less than 970 kB of memory.

Both CPUs in the machines with the two Intel CPUs have their own directly connected memory (visible in Fig. A.2 of Appendix A), and we made sure that all tool executions use only memory belonging to the same CPU as their CPU core(s), to avoid performance influence from NUMA. We also made sure that even the start of each tool execution already occurred on its assigned core, such that migration between cores, or even between CPUs, was avoided. There are no hardware caches that are shared between the CPUs in this system, and no swap space. There is also no correlation between the fact that a tool execution experienced a slow-down if both CPUs were active and the specific CPU that the tool was executed on. For both CPUs, there were tool executions with unaffected performance as well as tool executions with significantly increased CPU time.

The I/O (which would be a shared bottleneck) was kept to a minimum during benchmarking. Apart from loading the Java virtual machine and the verifier, each run used I/O only for loading the input file and for writing a log file with the output of the verifier (output of each tool execution was 15 kB on average and always less than 750 kB). Switching between a local SSD and a file server connected via network for all I/O except loading the JVM did not influence the performance, indicating that I/O indeed was not a bottleneck. We also verified that no additional temporary files were used. The tool executions were independent and the processes did not communicate with each other. Enabling Turbo Boost decreased the necessary CPU time in general and made the effect slightly larger. Upgrading the Linux kernel from version 3.13 to 3.19 improved the performance in general by approximately 1 to 2 %, but did not change the effect of decreased performance for using multiple CPUs. The machine is kept in an air-conditioned server room and did not throttle due to overheating during benchmarking.

For further investigation, we collected some statistics about the CPUs using the perf framework of the Linux kernel[5] while executing the tool. (These statistics are collected using internal hardware counters of the CPU and do not affect performance.) The result was that the number of context switches, CPU-migrations of threads, page faults, executed instructions and branches, and branch misses for one execution of the whole benchmark set were nearly the same, regardless of whether one or two CPUs were used, and thus gave no indication where the performance impact could come from.

One possible reason could be that Linux keeps only one global file-system cache that is shared for all processes from all CPUs. This means that, after a file has been loaded into the file-system cache, accessing the content of this file will be somewhat faster from one CPU than from the other CPU(s) due to NUMA. This behavior cannot be disabled. To avoid this effect, we used two separate on-disk copies of the benchmarked tool, and each copy was used for all runs executing on one specific CPU. Compared to using one on-disk copy of the benchmarked tool, this did not change the performance. While there was

[5] `https://perf.wiki.kernel.org`

still only a single install of the JVM used for all tool executions, we can assume that the single copy of the JVM in the file-system cache of the Linux kernel is not the reason for this effect. After all, the experiment from Sect. 11.6 showed a smaller effect even if *all* memory accesses were indirect, not only those to the JVM files in the file-system cache.

12. State-of-the-Art Benchmarking with Cgroups and Containers

We listed aspects that are mandatory for reliable benchmarking, and explained flaws of existing methods. In the following, we present two technologies that can be used to avoid these pitfalls.

12.1. Introducing Cgroups for Benchmarking

Control groups (cgroups) are a feature of the Linux kernel for managing processes and their resource usage, which is available since 2007 [218]. Differently from all other interfaces for these problems, cgroups provide mechanisms for managing groups of processes and their resources in an atomic and race-free manner, and are not limited to single processes. All running processes of a system are grouped in a hierarchical tree of cgroups, and most actions affect all processes within a specific cgroup. Cgroups can be created dynamically and processes can be moved between them. There exists a set of so-called *controllers* in the kernel, each of which controls and measures the consumption of a specific resource by the processes, within each cgroup. For example, there are controllers for measuring and limiting CPU time, memory consumption, and I/O bandwidth.

The cgroups hierarchy is made accessible to programs and users as a directory tree in a virtual file system, which is typically mounted at `/sys/fs/cgroups`. Usual file-system operations can be used to read and manipulate the cgroup hierarchy and to read resource measurements and configure limits for each of the controllers (via specific files in each cgroup directory). Thus, it is easy to use cgroups from any kind of tool, including shell scripts. Alternatively, one can use a library such as `libcg`[1], which provides an API for accessing and manipulating

[1] `http://libcg.sourceforge.net`

the cgroup hierarchy. Settings for file permission and ownership can be used to fine-tune who is able to manipulate the cgroup hierarchy.

When a new process is started, it inherits the current cgroup from its parent process. The only way to change the cgroup of a process is direct access to the cgroup virtual file system, which can be prevented using basic file-system permissions. Any other action of the process, whether changing the POSIX process group, detaching from its parent, etc., will not change the cgroup. Thus, cgroups can be used to reliably track the set of (transitive) child processes of any given process by putting this process into its own cgroup. We refer to the manual for details.[2]

The following cgroup controllers are relevant for reliable benchmarking:

`cpuacct` measures, for each cgroup, the accumulated CPU time that is consumed by all processes of the cgroup. A time limit cannot be defined, but can be implemented in the benchmarking environment by periodically checking the accumulated time.

`cpuset` supports restricting the processes in a cgroup to a subset of the available CPU cores. On systems with more than one CPU and NUMA, it allows restricting the processes to specific parts of the physical memory. These restrictions are applied additionally to those set with `sched_setaffinity`, such that changes to the latter will not affect restrictions made via cgroups.

`freezer` supports freezing all processes of a cgroup in a single operation. This can be used for reliable termination of a group of processes by freezing them first, sending the kill signal to all of them, and afterwards unfreezing ("thawing") them. This way the processes do not have the chance to start other processes because between the time the first and the last process receive the kill signal, none of them can execute anything.

`memory` supports, for each cgroup, restricting maximum memory usage of all processes together in the cgroup, and measuring current and peak memory consumption. If the defined memory limit is reached by the processes in a

[2] `https://www.kernel.org/doc/Documentation/cgroup-v1`

cgroup, the kernel first tries to free some internal caches that it holds for these processes (for example disk caches), and then terminates at least one process. Alternatively, instead of terminating processes, the kernel can send an event to a registered process, which the benchmarking framework can use to terminate all processes within the cgroup. The kernel counts only actually used pages towards the memory usage, and because the accounting is done per memory page, shared memory is handled correctly (every page that the processes use is counted exactly once).

The `memory` controller supports two limits for memory usage, one on the amount of physical memory that the processes can use, and one on the amount of physical memory plus swap memory. If the system has swap memory, both limits need to be set to the same value for reliable benchmarking. If only the former limit is set to a specific value, the processes could use so much memory plus all of the available swap memory (and the kernel would automatically start swapping out the processes if the limit on physical memory is reached). Similarly, for reading the peak memory consumption, the value of physical memory plus swap memory should be used. Sometimes, the current memory consumption of a cgroup is not zero even after all processes of the cgroup have been terminated, if the kernel decided to still keep some pages of these processes in its disk cache. To avoid influencing the measurements of later tool executions by this, a cgroup should be used only for a single run and deleted afterwards, with a new tool execution getting a new, fresh cgroup.[3]

As described in Sect. 10.4, the Linux kernel does not use a consistent scheme for assigning processor ids to virtual cores and node ids to memory regions, which are the ids used by the `cpuset` controller. Information about the hardware topology and the relations of CPU cores to each other and to memory regions needs to be read from the directory tree `/sys/devices/system/cpu/`.[4] There, one can find a directory named `cpu<`*i*`>` for each virtual core *i*, and inside each such directory, the following information is present: the symlinks named `node<`*j*`>` point to the NUMA region(s) of this virtual core, `topology/physical_package_id` contains the physical id of this virtual core, `topology/core_id` contains the core id of this virtual core, `topology/core_siblings_list` contains the

[3] Or clear the caches with `drop_caches`.
[4] Or use a library that does this reliably.

virtual cores of the same CPU as this virtual core, and the file `topology/thread_siblings_list` contains the virtual cores of the same physical core as this virtual core.

12.2. Benchmarking Containers Based on Namespaces

Container is a common name for an instance of OS-level virtualization on Linux. Contrary to virtual machines, there is no virtual hardware simulated for a container, and a container does not have its own kernel running. Instead, the applications in a container run directly on the same kernel as applications outside the container, without any additional layers that would reduce performance. However, the kernel provides a limited view of the system to processes inside a container, such that these processes are restricted in what they can do with regard to the system outside their container. Containers can be used to execute a single application in isolation. A well-known framework for creating containers in Linux is Docker.

The key technology behind containers are *namespaces*[5], which are a feature of the Linux kernel for providing individual processes or groups of processes with a different view on certain system resources compared to other processes. There exist different kinds of namespaces, which can be used individually or in combination, each responsible for isolating some specific system resources. For example, assigning a different network namespace to a process will change which network interfaces, IP addresses, etc., the process sees and is able to use. Assigning a different namespace for process ids (PIDs) will change which processes can be seen, and which PIDs they seem to have.

Using namespaces, we can create benchmarking containers that prevent any communication or interference with processes outside the container. We need only a modern ($\geq$ 3.8) Linux kernel for this; no other software is necessary. The benchmarked tool will typically not notice that it is executed in such a container. The performance of executing a process within a separate namespace (i.e., inside a container) is comparable to executing it directly in the initial namespace because

[5] `http://man7.org/linux/man-pages/man7/namespaces.7.html`

the process still interacts directly with the same kernel and there are no additional layers like in hardware-virtualization solutions.

Since Linux 3.8 (released February 2013) it is possible to create and configure namespaces as a regular user without additional permissions. At a first glance it may seem that creating and joining a namespace can give a process more permissions than it previously had (such as changing network configuration or file-system mounts), but all these new permissions are only valid inside the namespace, and none of these actions affect the system outside of this namespace.

The Linux kernel provides the following namespaces that are relevant for reliable benchmarking:

`mount` namespaces allow changing how the file-system layout appears to the processes inside the namespace. Existing directories can be mounted into a different place using "bind" mounts (similar to symbolic links), and mount points can be set read-only. Unprivileged users cannot create new mounts of most file systems, even if they are in a mount namespace, but for a few special file systems this is allowed. For example, RAM disks can be mounted by everyone in a mount namespace.

`ipc` namespaces provide separation of several different forms of interprocess communication (IPC), such as POSIX message queues.

`net` namespaces isolate the available network interfaces and their configuration, e.g., their IP addresses. By default, a new network namespace has only a loopback interface with the IP addresses `127.0.0.1` (IPv4) and `::1` (IPv6) and thus, processes in such a namespace have no access to external network communication, but can still use the loopback interface for communication between the processes within the same namespace (note that the loopback interfaces of different network namespaces are different: a loopback interface cannot be used for communication with processes of separate network namespaces).

`pid` namespaces, which can be nested, provide a separate range of process IDs (PIDs), such that a process can have different PIDs, one in each (nested) namespace that it is part of. Furthermore, the PID namespace also affects mounts of the `/proc` file system, which is the place where the Linux kernel makes information available about the currently running processes. If a new `/proc` file system is mounted in a PID namespace, it will list only those processes that are part of this namespace. Thus, a PID namespace can be used to restrict which other processes a process can see: only such processes that are in the same PID namespace are visible. This prevents for example sending signals to processes and killing them, if they are not visible in the current namespace. Additionally, because the first process in every new PID namespace has a special role (like the `init` process of the whole system), the kernel automatically terminates all other processes in the namespace when this first process terminates. This can be used for a reliable cleanup of all child processes.

`user` namespaces provide a mapping of user and group ids, such that a certain user id inside a namespace appears as a different id outside of it. Creating a user namespace is necessary for regular (non-`root`) users in order to create any other kind of namespace.

For each kind of namespace, an initial namespace is created at boot. These are the namespaces that are used for all processes on most systems. Processes can create new namespaces and move between them, but for the latter they need to get a reference to the target namespace, which can be prevented by using the PID namespace. Thus, it is possible to prevent processes from escaping back into an existing namespace such as the initial namespace. Furthermore, all namespace features are implemented in a way such that it is not possible to escape the restrictions of a namespace by creating and joining a fresh namespace: a freshly created namespace will not allow viewing or manipulating more system resources than its parent.

With mount namespaces we can customize the file-system layout. We can for example provide an own private `/tmp` directory for each tool execution by binding a freshly created directory to `/tmp` in the container of the tool execution. This avoids any interference between tool executions due to files in `/tmp`, and the same solution can also be applied for all other directories whose actual content should be invisible in the container. In order to ensure that no files created by a tool execution are left over in other directories (and possibly influence later tool executions), we can set all other mount points to read-only in the container. However, this can be inconvenient, for example, if the tool should produce some output file, or if it expects to be able to write in some fixed directory (like the home directory). The solution for this is to use an overlay file system.

Overlay file systems[6] use two existing directories, and appear to layer one directory over the other. Looking at the overlay file system, a file in the upper layer shadows a file with the same name in the lower layer. All writes done to the overlay file system go to the upper layer, never to the lower layer, which is guaranteed to remain unchanged. We can use this to present a directory hierarchy to the benchmarked tool that is initially equal to the directory hierarchy of the system and appears to be writable as usual, but all write accesses are rerouted to a temporary directory and do not affect the system directory. To do so we mount an overlay file system with the regular directory hierarchy of the system as lower layer and an empty temporary directory as upper layer.

An overlay file system is available in the Linux kernel since version 3.18, and on Ubuntu it can be used by regular users inside a mount namespace. On other distributions or older kernels we have to fall back to either read-only mounts or giving direct write access to the file-system at least for some directories.

[6] https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt

13. BenchExec: A Framework for Reliable Benchmarking

In the following, we describe our implementation BenchExec, a benchmarking framework for Linux that fulfills the requirements from Chapter 9 by using the techniques of cgroups and namespaces from Chapter 12. It is available on GitHub[1] as open source under the Apache 2.0 license.

BenchExec consists of two parts, both written in Python. The first part is `runexec`, responsible for benchmarking a single run of a given tool in isolation, including the reliable limitation and accurate measurement of resources, and encapsulates the use of cgroups and namespaces. This part is designed such that it is easy to use also from within other benchmarking frameworks. The second part is responsible for benchmarking a whole set of runs, i.e., executing one or more tools on a collection of input files by delegating each run to the first part and then aggregating the results. It consists of a program `benchexec` for the actual benchmarking and a program `table-generator` for postprocessing of the results.

13.1. System Requirements

Full support for all features of BenchExec is available on Ubuntu with a Linux kernel of at least version 3.18 (default since Ubuntu 15.04). On older kernels or other distributions, the overlay file system cannot be used and the file-system layout inside the containers needs to be configured differently. On kernels older than Linux 3.8, BenchExec's use of containers needs to be disabled completely and thus isolation of runs will not be available, but other features of BenchExec

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

(such has accurate resource measurements and limits, and sensible allocation of hardware resources) remain usable.

For the use of cgroups by BenchExec, a few requirements are necessary that may demand assistance by the administrator of the benchmarking machine. Cgroups including the four controllers listed in Sect. 12.1 must be enabled on the system and the account for the benchmarking user needs the permissions to manipulate (a part of) the cgroup hierarchy. If the benchmarking machine has swap, swap accounting must be enabled for the `memory` controller. For enabling cgroups and giving permissions, we refer to standard Linux documentation. For more details on how to setup the prerequisites for BenchExec, we refer to the respective chapter of the documentation.[2]

After these steps, no further root access is necessary and everything can be done with a normal user account. Thus, it is possible to use machines for benchmarking that are not under one's own administrative control. By creating a special cgroup for benchmarking and granting permissions only for this cgroup, it is also possible for the administrator to prevent the benchmarking user from interfering with other processes and to restrict the total amount of resources that the benchmarking user may use. For example, one can specify that a user may use only a specific subset of CPU cores and amount of memory for benchmarking, or partition the resources of shared machines among several users.

13.2. Benchmarking a Single Run

We define a *run* as a single execution of a tool, with the following input:

- the full command line, i.e., the path to the executable with all arguments, and optionally,
- the content supplied to the tool via `stdin`,
- the limits for CPU time, wall time, and memory, and
- the list of CPU cores and memory banks to use.

A run produces the following output:

[2] `https://github.com/sosy-lab/benchexec/blob/master/doc/INSTALL.md`

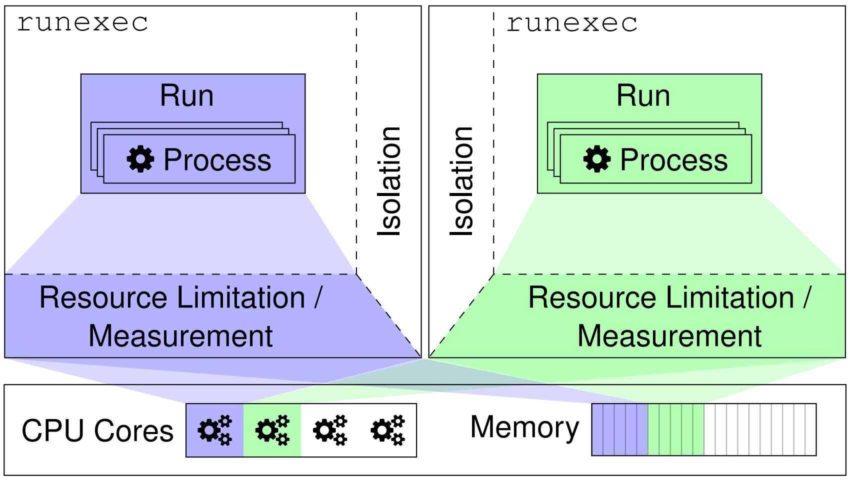


Figure 13.1.: Resource control and process isolation by `runexec`; a run can consist of many processes; the vertical bars "Isolation" illustrate that each run is executed in isolation and protected to not access other runs; the horizontal bars "Resource Limitation / Measurement" illustrate that the resources are controlled by cgroups and a run can access only the explicitly assigned resources

- measurement values (e.g., CPU time, wall time, and peak memory consumption of the tool),
- the exit code of the main process,
- output written to `stdout` and `stderr` by the tool, and
- the files created or written by the tool.

The program `runexec` executes a tool with the given input, provides the output of the run, and ensures (using cgroups and namespaces) adherence to the specified resource limits, accurate measurement of the resource usage, isolation of the process with regard to network usage, signals, and file-system writes, and reliable cleanup of processes after execution (i.e., no process survives). The benchmarking containers created by `runexec` are illustrated in Fig. 13.1. If necessary, the benchmarking containers can be customized via additional options, e.g., with regard to the file-system layout (which directories are hidden in the container or made read-only etc.), or whether network access is allowed from within the container.

If `runexec` is used as stand-alone tool, the inputs are passed to `runexec` as command-line parameters. Alternatively, `runexec` can be used as a Python module for a more convenient integration into other Python-based benchmarking frameworks.

An example command line for executing a tool on all 16 (virtual) cores of the first CPU of a dual-CPU system, with a memory limit of 16 GB on the first memory bank, and a time limit of 100 s is:

```
runexec --timelimit 100s --memlimit 16GB
        --cores 0-7,16-23 --memoryNodes 0 --
        <TOOL_CMD>
```

The output of `runexec` then looks as follows (log on `stderr`, result on `stdout`):

```
10:35:35 - INFO - Starting command <TOOL_CMD>
10:35:35 - INFO - Writing output to output.log
exitcode=0
returnvalue=0
walltime=1.51596093178s
cputime=2.514290687s
memory=130310144
```

In this case, the run took 1.5 s of wall time, and the tool used 2.5 s of CPU time and 130 MB of RAM before returning successfully (return value 0). The same could be achieved by importing `runexec` as a module from within a Python program with a code snippet as in Listing B.1 of Appendix B, which returns a dictionary that holds the same information as the key-value pairs printed to `stdout` in the example above. The precise meaning of each of these values is explained in the BenchExec documentation.[3]

13.3. Benchmarking a Set of Runs

Benchmarking typically consists of processing runs on hundreds or thousands of input files, and there may be several different tools or several configurations of the same tool that run on the same input files. BenchExec provides two programs that allow to perform such large experiments and analyze the results as easily as possible. An overview over the process of using these programs can be seen in Fig. 13.2.

[3] `https://github.com/sosy-lab/benchexec/blob/master/doc/run-results.md`

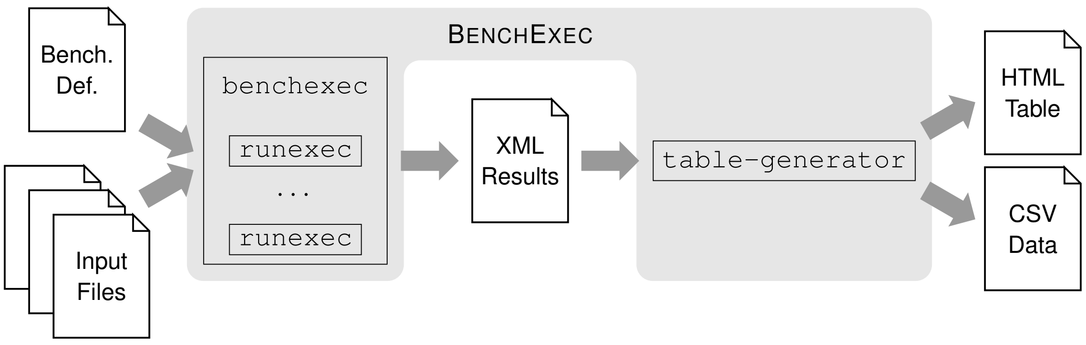


Figure 13.2.: Benchmarking with `benchexec`; `benchexec` supports the user by automating the execution of experiments with many runs; besides the benchmarked tool, it expects the benchmark definition and a set of input files; `benchexec` executes a series of runs, which are handled (each separately) by `runexec` (cf. Fig. 13.1); the result is an XML file that contains the raw result data; convenient postprocessing is possible using `table-generator`, which creates customized CSV files and data visualizations (tables and plots) based on HTML

`benchexec` is a program that executes a set of runs. It receives as input a benchmark definition, which consists of the following components:

- a set of input files,
- the name of the tool to use,
- command-line arguments for the tool (e.g., to specify a tool configuration),
- the limits for CPU time, memory, and number of CPU cores, and
- the number of runs that should be executed in parallel.

This benchmark definition is given in XML format; an example is available in the tool documentation[4] and in Listing B.2 of Appendix B. Additionally, a tool-info module (a tool-specific Python module) needs to be written that contains functions for creating a command-line string for a run (including input file and user-defined command-line arguments) and for determining the result from the exit code and the output of the tool. Such a tool-info module typically has under 50 lines of Python code, and needs to be written only once per tool. Experience shows that

[4] `https://github.com/sosy-lab/benchexec/blob/master/doc/benchexec.md`

this tool-info module can be written and integrated successfully into BenchExec also by developers that previously were not familiar with BenchExec.[5]

We are often also interested in classifying the result into expected and incorrect answers. BenchExec supports this for SMT solvers that are compliant to the SMT-LIB standard [16], and for the domain of automatic software verification, where it gets as input a property to be verified in the format used by SV-COMP [22] [6]. Classification of results for further domains can be added with a few lines of code.

`benchexec` and its benchmark-definition format also support specifying different configuration options for subsets of the input files, as well as several different tool configurations at once, each of which will be benchmarked against all input files (cf. Listing B.2 of Appendix B).

The program `benchexec` starts with trying to find a suitable allocation of the available resources (CPU cores and memory) to the number of parallel runs. For this, it first checks whether there are enough CPU cores and memory in the system to satisfy the core and memory requirements for all parallel runs. Then it assigns cores to each parallel run such that (if possible) a single run is not spread across different CPUs, and different runs use different CPUs or at least different physical cores. Measurement problems due to NUMA are avoided by letting each run use only memory that is directly connected to the CPU(s) on which the run is scheduled. Thus, `benchexec` automatically guarantees the best allocation of the available resources that minimizes the nondeterministic performance influences that were shown in Chapter 11 as far as possible for the given number of parallel runs.

Afterwards, `benchexec` uses `runexec` to execute the benchmarked tool on each input file with the appropriate command line, resource limits, etc. It also interprets the output of the tool according to the tool-info module (if applicable, it also determines whether the result was correct). The result of `benchexec` is a data set (in XML format) that contains all information from the runs: tool result, exit code, and measurement values. The output of the tool for each run is available in separate files. Additional information such as current date and time, the host

[5] SV-COMP'16 for the first time required all participating teams to contribute such a module for their tool to BenchExec [23], leading to 21 new tools being integrated into BenchExec.

[6] Tools that do not support this format can also be benchmarked. In this case, the property is not passed to the tool, but used only internally by BenchExec to determine the expected result.

and its system information (CPU and RAM), values of environment variables, and the effective resource limits and hardware allocation are also recorded.

`table-generator` is a program that produces tables from the results of one or more executions of `benchexec`. If results of several executions are given to `table-generator`, they are combined and presented one per column group in the table, allowing to easily compare the results, for example, across different configurations or revisions of a tool, or across different tools. Each line of the generated table contains the results for one input file. There are columns for the tool result and measurement values (such as CPU time, wall time, memory usage, etc.). These tables are written in two formats. A CSV-based format allows further processing of the raw data, such as with gnuplot or R for producing plots and statistical evaluations, a spreadsheet program, or LaTeX packages for reading CSV files in order to present results in a paper. The second format is HTML, which allows the user to view the tables conveniently with nothing more than a web browser. The HTML table additionally provides access to the text output of the tool for each run and contains aggregate statistics and further relevant information such as tool versions, resource limits, etc. Furthermore, the HTML table is interactive and supports filtering of columns and rows. HTML tables produced by `table-generator` even allow generating scatter and quantile plots on-the-fly for selected columns upon user request. Examples of such tables can be found online.[7]

If a tool outputs further interesting data (e.g., for a verifier, this could be time statistics for individual parts of the analysis, number of created abstract states, or SMT queries), those data can also be added to the generated tables if a function is added to the tool-specific Python module which extracts such data values from the output of the tool. All features of the table (such as generating plots) are immediately available for the columns with such data values as well.

13.4. Comparison with Requirements for Reliable Benchmarking

BenchExec fulfills all requirements for reliable benchmarking from Chapter 9. By using the kernel features for resource management with cgroups that are

[7] `https://www.sosy-lab.org/research/benchmarking/#tables`

described in Sect. 12.1, BenchExec ensures accurate measuring and limiting of resources for sets of processes (Sect. 9.1), as well as reliable termination of processes (Sect. 9.2). The algorithms for resource assignment that are implemented in `benchexec` follow the rules of Sects. 9.3 and 9.4, and assign resources to runs such that mutual influences are minimized as far as possible on the given machine, while warning the user if a meaningful resource allocation cannot be done (e.g., if attempting to execute more runs than physical cores are available, or scheduling multiple runs on a CPU with active Turbo Boost). Note that because `runexec` alone handles only single runs, it cannot enforce a proper resource allocation across multiple runs, and users need to use either `benchexec`, an own benchmarking framework, or manual interaction for proper resource allocation across multiple runs. For swapping (cf. Sect. 9.5), BenchExec does what is possible without root privileges: it enforces that swap memory is also limited and measured if present, monitors swap usage, and warns the user if swapping occurs during benchmarking.

Isolation of runs as described in Sect. 9.6 is implemented in BenchExec by executing each run in a container with fresh namespaces. The PID namespace prevents the tool from sending signals to other processes, and network and IPC namespaces prevent communication with other processes and systems. The mount namespace allows customizing the file-system layout. This is used to provide a separate `/tmp` directory for each run, as well as for transparently redirecting all write accesses such that they do not have any effect outside the container. If necessary, the user can choose to weaken the isolation by allowing network access or write access to some directories. Of course, it is then up to the user to decide whether the incomplete isolation of runs will hinder reliable benchmarking.

Additionally, note that while BenchExec cannot ensure that users always present benchmark results correctly (e.g., in publications), BenchExec follows standard recommendations for valid presentation of results, e.g., it always uses SI units and rounds values to a fixed number of significant digits instead of decimal places.

13.5. Discussion

We would like to discuss a few of the design decisions and goals of BenchExec.

BenchExec aims at not impacting the external validity of experiments by avoiding to use an overly artificial environment (such as a virtual machine) or influencing the benchmarked tool in any way (except for the specified resource limits and the isolation). For example, while BenchExec allocates resources such as CPU cores for runs to avoid influences between parallel runs, it does not interfere with how the assigned resources are used *inside* each run. If tools with multiple threads or processes are benchmarked, an appropriate resource allocation between these threads and processes within the limits of the run remains the responsibility of the user. The use of namespaces by BenchExec allows us to execute the tool as if it were the only program on a machine with no network connection. Both cgroups and namespaces are present and active on a standard Linux machine anyway, and their use does not add significant overhead.

We designed BenchExec with extensibility and flexibility in mind. Support for new tools and result classifications can be provided by adding a few lines of Python code. The program `runexec`, which controls and measures resources of the actual tool execution, can be used separately as a stand-alone tool or a Python module, for example within other benchmarking frameworks. Result data are present as CSV tables, which allows processing with standard software.[8]

We chose not to base BenchExec on an existing container framework such as LXC or Docker because, while these provide resource limitation and isolation, they typically do not focus on benchmarking, and a fine-grained controlling of resource allocation as well as measuring resource consumption may be difficult or impossible. Furthermore, requiring a container framework to be installed would significantly limit the amount of machines on which BenchExec can be used, for example, because on many machines (especially in clusters for high-performance computing) the Linux kernel is too old, or such an installation is not possible due to administrative restrictions. Using cgroups and namespaces directly minimizes the necessary version requirements, the installation effort, and the necessary access rights, and makes it easy to use a fallback implementation without containers but still with reliable resource measurements due to cgroups if

[8] For example, BenchExec is used to automatically check for regressions in the integration test-suite of CPAchecker.

necessary.[9] For example, using Docker would require to give root-level access for the benchmarking machine to all users who want to execute benchmarks.[10] All features of Docker that are necessary for reliable benchmarking (such as process isolation) are available in BenchExec as well and are implemented in a way that after a one-time setup, they only need a few specific privileges that can be granted individually and without a security risk.

We use XML as input and output format because it is a structured format that is readable and writable by both humans and tools, and it is self-documenting. Since XML supports comments, users can also use comments in the benchmark-specification and table-definition files to document their setup. We can store customized result data, but also additional meta data in the result file. This allows documenting information about the benchmarking environment, which is important in scientific work because it increases the replicability and trust of the results.

Python was chosen as programming language because it is expected to be available on practically every Linux machine, and it makes it easy to write the tool-specific module even for people that do not have much experience in programming.

For the future, several research directions and extensions of BenchExec are possible. If necessary, one could work on lifting some of the stated restrictions, and, for example, implement techniques for reliable benchmarking of tools with heavy I/O or use of GPU resources. BenchExec relies on the Linux kernel for the precision of its measurements. To rule out remaining potential nondeterministic effects such as the layout of memory allocations, some users might want to increase the precision by running the same experiment repeatedly. This needs to be done manually today, but of course built-in support could be added. Furthermore, we are interested in measuring the energy consumption of benchmarks.

[9] We successfully use BenchExec on four different clusters, each under different administrative control and with software as old as SuSE Enterprise 11 and Linux 3.0, and on the machines of the student computer pool of our department.

[10] cf. `https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface`

13.6. Encouraging Replicable Experiments

Reliable benchmarking results obtained by accurate and precise measurements are necessary but not sufficient for replicable experiments. The experimental setup also needs to be documented precisely enough that the experiment can be performed again by different people. While careful documentation is ultimately the responsibility of the experimenter, BenchExec is designed such that it encourages and facilitates documentation. For example, BenchExec accepts the benchmark definition with all relevant properties in a text file (XML-based), which can be explained using comments, and which can be easily archived using a version-control system. This strategy is used, for example, by SV-COMP since 2016[11] [23]. BenchExec is also able to automatically store benchmark results in a Git repository. The result files do not only contain the raw measurement values, but BenchExec also documents important information about the experimental setup, including the hardware characteristics (CPU model, RAM size), the operating-system version, values of environment variables, the version number of the benchmarked tool (if available), the resource limits, and the full command-line of the executed run. Taken together, these features conveniently allow to document everything related to a series of experiments for example in a Git repository, which could be shared with collaborators, or even reviewers or the general public.

[11] Cf. SV-COMP benchmark definitions at `https://github.com/sosy-lab/sv-comp`

14. Related Work

Replicable experiments are an important topic in computer science in general. For example, the ACM has developed a policy on "Result and Artifact Review and Badging" [1] with the goal of standardizing artifact evaluation, and labels articles that support experiment replication with special badges. This work was inspired by effort in this direction in the SIGPLAN and SIGSOFT communities.[2]

Besides the issues that we discussed in this thesis, there are more sources of nondeterministic effects that may influence performance measurements in our target domain. For example, the memory layout of the process can affect performance due to differences in cache alignment and cache hit rates [102], and the memory layout can be influenced by factors such as symbol names [130, 160], environment variables and order of objects during linking [198], and nondeterministic memory-allocation strategies of the OS [142].

Benchmarking solutions are also developed in other communities. For example, the Mininet Hi-Fi project [135] provides a framework for experiments in computer networking. It emulates a virtual network on a single machine by using cgroups and namespaces to setup and control a set of virtual hosts and network connections between them.

14.1. Benchmarking Strategies

Several strategies have been proposed for dealing with nondeterministic effects that influence measurements by executing runs several times. The Execution-Time-Measurement Protocol (EMP) [223] was designed for improving the precision of time measurements by using a sequence of repeated executions and measurements, for example for avoiding the influence of background processes of the

[1] `https://www.acm.org/publications/policies/artifact-review-badging`
[2] `http://evaluate.inf.usi.ch/artifacts/aea`

operating system. Furthermore, the authors of EMP also identified other sources of measurement errors that the administrator of the benchmarking machine should address. For example, time measurements (even for CPU time) can be imprecise on machines without time synchronization via NTP.

The authors of DataMill [203] propose to make benchmarking more reliable by explicitly varying as many hardware and software factors as possible in a controlled manner while benchmarking, e.g., the hardware architecture, CPU model, memory size, link order, etc. To do so, they rely on a diverse set of worker machines, which are rebooted for each benchmark run into a specific OS installation.

In general, these strategies focus on ruling out nondeterministic measurement noise, but do not handle some of the sources of measurement errors identified in this work, such as the CPU-time measurement of tools with several processes, and may thus still produce arbitrarily large measurement errors. On the other hand, BenchExec aims at ruling out such qualitative measurement errors, and relies on the Linux kernel for precise and accurate measurements, but does not increase the precision in case of nondeterministic effects that it cannot control, such as memory-allocation strategies. If such effects are present, users can of course increase the precision by executing benchmarks several times with BenchExec and aggregating the results. In the future, benchmarking with cgroups and namespaces could (and should) be combined with one of the strategies described above to address the respective problem. BenchExec can serve as low-level measurement tool in a distributed framework such as DataMill, or by implementing EMP on top of BenchExec.

14.2. Benchmarking Tools

In the verification community, there exist several benchmarking tools that have the same intent as our benchmarking framework. However, as of June 2017, no tool we investigated fulfills all requirements for reliable benchmarking, which are presented in Chapter 9. In the following, we discuss several existing benchmarking tools in their latest versions as of June 2017. Our selection is not exhaustive, because there exist many such tools.

The tool RunLim[3], in version 1.10, allows benchmarking an executable and limits both CPU time and memory. It samples time and memory consumption recursively for a process hierarchy, and thus cannot guarantee accurate measurements and limit enforcement. The tool cannot terminate a process hierarchy reliably, because it only terminates the main process with `kill`. The tool pyrunlim[4], a port of RunLim to the Python programming language, has a few more features, such as setting the CPU affinity of a process, and aims at killing process hierarchies more reliably. However, in the latest version 2.15[5], it does not use cgroups and also takes sample measurements recursively over process hierarchies, which —like all sampling-based methods— is not accurate.

The CProver Benchmarking Toolkit (CPBM)[6], available in version 0.5, ships helpers for verification-task patch management and result evaluation, and also supports benchmarking. However, the limits for CPU time and memory are enforced by `ulimit`[7], and thus, the benchmarking is not accurate.

Furthermore, none of the tools mentioned so far attempts to isolate independent benchmark runs from each other. This could be done in addition to using one of these tools, e.g., by executing each run under a fresh user account. This would require a substantial amount of additional implementation effort.

The Satisfiability Modulo Theories Execution Service (SMT-Exec)[8] was a solver execution service provided by the SMT-LIB initiative. For enforcing resource limits, SMT-Exec used the tool TreeLimitedRun[9], which uses the system calls `wait` and `setrlimit`, and thus, is prone to the restrictions explained in Chapter 10.

StarExec [222], a web-based service developed at the Universities of Iowa and Miami, is the successor of SMT-Exec. The main goal of StarExec is to facilitate the execution of logic solvers. The Oracle Grid Engine takes care of queuing and scheduling runs. For measuring CPU time and memory consumption, as well as enforcing resource limits, StarExec delegates to runsolver[10] [209], available in version 3.3.5, which also is prone to the limitations from Chapter 10.

[3] `http://fmv.jku.at/runlim`
[4] `http://alviano.net/2014/02/26`
[5] Git revision `b9b2f11` from 2017-05-02 on `https://github.com/alviano/python/tree/master/pyrunlim`
[6] `http://www.cprover.org/software/benchmarks`
[7] cf. `verify.sh` in the CPBM package
[8] `http://smt-exec.org`
[9] `http://smtexec.cs.uiowa.edu/TreeLimitedRun.c`
[10] `http://www.cril.univ-artois.fr/~roussel/runsolver`

The Versioning Competition Workflow Compiler (VCWC) [74] is an effort to create a fault-tolerant competition platform that supports competition maintainers in order to minimize their amount of manual work. This project, in the latest development version [11], defines its own benchmarking container, and uses `sudo` and `schroot` [12] for executing each run in a fresh environment. However, the setup for this needs root access to the machine, and all parallel runs are executed under the same user account, meaning that an executed process can still see and influence processes of parallel runs. Furthermore, VCWC relies on `ulimit` to enforce time limits. If the administrator of the benchmarking machine manually designed and created a cgroup hierarchy that enforces an appropriate partitioning of CPU cores and memory nodes, and defined a memory limit, VCWC can execute runs within these existing cgroups, but it cannot automatically create the appropriate cgroups, as BenchExec does. Furthermore, measurement of CPU time and memory, as well as termination of processes, is not implemented with cgroups, and hence, may fail.

The tool BenchKit [13] [170], available in version 1.1, is also used for competitions, where participants submit a virtual-machine (VM) image with their tool and all necessary software. BenchKit executes the tool within an instance of this VM and measures the resource usage of the tool and the OS together in the VM. Our framework executes all tools natively on the host system and allows precise measurement of the resource consumption of the tool in isolation, without influence from factors such as the VM's OS. BenchKit measures CPU time and memory consumption of the VM using sampling with the performance monitoring tool sysstat [14], which is available in version 11.4.4. BenchKit does not ensure that the CPU cores and the memory for a run are assigned such that hyperthreading and NUMA are respected. For each single run with BenchKit, i.e., each pair of tool and input file, a new VM has to be booted, which on average takes 40 s to complete [170]. Execution of a tool inside a VM can also be slower than directly on the host machine. Our approach based on cgroups and containers has a similar effect of isolating the individual runs and their resource usage, but comes at practically zero overhead. Our tool implementation was

[11] Git revision `9d58031` from 2013-09-13 on `https://github.com/tkren/vcwc`
[12] A utility for executing commands in a chroot environment, cf. `http://linux.die.net/man/1/schroot`
[13] `http://www.cosyverif.org/benchkit.php`
[14] `http://sebastien.godard.pagesperso-orange.fr`

successfully used in SV-COMP'17, which consisted of more than 400 000 runs, plus an uncounted number of runs for testing the setups of the participating tools [24]. An overhead of 40 s per run would have required additional 190 CPU days for the main competition run alone, a prohibitive overhead.

Part III.

Experimental Evaluation

Contents

15. Experiment Setup **153**

15.1. Verification Tasks 154
15.2. Tool Versions 154
15.3. Configurations of the Predicate CPA 154
15.4. CPACHECKER Configuration 155
15.5. Hardware and Software Environment 156
15.6. Replicability and Availability of Results 157
15.7. Presentation of Results 157

16. Comparison of SMT Solvers and Theories **159**

16.1. Bitprecise Theories 162
16.2. Theories with Linear Arithmetic 167
16.3. Comparison of Bitprecise and Linear Theories 171
16.4. Conclusions 174

17. Comparison with State of the Art **175**

15. Experiment Setup

In the following, we evaluate our unifying framework for predicate analysis from Part I using the techniques for reliable benchmarking from Part II. The goal of this evaluation is to show the usefulness of our framework for predicate analysis, i.e., that its implementation is effective and efficient and that its high configurability and flexibility allows us to easily perform experimental studies that provide new and valuable insights.

One such experimental study that is based on and was made possible by our framework for predicate analysis is an already published detailed comparison of the verification approaches BMC, *k*-induction, Impact, and predicate abstraction [28]. A preliminary version of our framework for predicate analysis had already enabled an in-depth comparison of Impact and predicate abstraction that provided insights about the key advantages of the two approaches [52]. Thus, we do not perform a study about comparing verification approaches here, but instead exploit the flexibility of our framework for predicate analysis by performing a study about the influence of SMT solvers and SMT theories across the four algorithms that we have expressed in our framework. This study was not possible before, because it requires a framework that allows combining all algorithms with all SMT solvers and various SMT theories while at the same time keeping all other influencing factors constant. In total, for this study we evaluated 120 different configurations with a total of 671 280 verification runs, which needed 3 620 days of CPU time.

In order to evaluate the effectivity and efficiency of our implementation, we compare it with several of the most successful verifiers of the 6th International Competition on Software Verification (SV-COMP'17) [24], which represent the top of the state of the art. Furthermore, note that our implementation in its configuration for predicate abstraction won four medals (including a gold medal) in SV-COMP'12 [19, 187] and contributed to winning another 40 medals in six years of SV-COMP (details in Sect. 1.1.2).

15.1. Verification Tasks

The set of verification tasks was taken from SV-COMP'17 [24]. We used only verification tasks where the property to verify is the reachability of a program location (excluding the properties for memory safety, overflows, and termination, which are not in our scope). We excluded the categories for the verification of recursive (*ReachSafety-Recursive*) and concurrent programs (*ConcurrencySafety*), which are not supported by the Predicate CPA. The resulting set of categories consists of the subcategory *DeviceDriversLinux64* of the category *SoftwareSystems*, and the following subcategories of the category *ReachSafety*: *Arrays*, *Bitvectors*, *ControlFlow*, *ECA*, *Floats*, *Heap*, *Loops*, *ProductLines*, and *Sequentialized*.

In total, our benchmark set contains 5 594 verification tasks, out of which 1 444 tasks contain a known specification violation, and 4 150 tasks are considered safe.

15.2. Tool Versions

The experiments using CPAchecker were executed with revision 24 567 of the branch `predicatecpa-evaluation`. All code of this branch is also integrated into the trunk of the project repository.[1] This revision of CPAchecker includes the latest revisions (as of January 10th, 2017) of the four integrated SMT solvers: MathSAT5 5.3.14, Princess 2016-12-26, SMTInterpol 2.1-327-g92cafef[2], and Z3 4.5.0-164-g8047f0d[3].

15.3. Configurations of the Predicate CPA

We are interested in evaluating our framework for predicate analysis in various configurations. Due to its high flexibility and configurability, it is impossible to evaluate all configurations, so we pick the following four, which resemble state-of-the-art approaches for software verification:

[1] Revision 24 567 of the branch corresponds to revision 24 091 of the trunk with the additional bug fixes from revisions 24 551 to 24 565 applied.

[2] While not an official release, this version was published by the developers via Maven Central (`https://search.maven.org/#search|gav|1|a:"smtinterpol"`).

[3] The latest official release 4.5 had a bug that was fixed by the Z3 developers for us, so we used the Git revision with the fix.

- BMC (cf. Sect. 5.2.1), with an initial loop bound of 1, an increment function $\mathsf{inc}(k) = k + 1$, and no maximal loop bound,
- k-induction (cf. Sect. 5.2.2), with an initial loop bound of 1, an increment function $\mathsf{inc}(k) = k + 1$, no maximal loop bound, and a continuously-refining generator for auxiliary invariants that uses an abstract domain of disjunctions of intervals [32],
- Impact with adjustable block-encoding (cf. Sects. 5.2.4 and 5.4.1), with ABE blocks ending at loop heads (blk^l) and interpolation-based forced covering ($\mathsf{fcover}^{\text{Impact}}$), and
- lazy predicate abstraction (cf. Sect. 5.2.3), with ABE blocks ending at loop heads (blk^l), boolean predicate abstraction, and interpolation as source of abstract facts for refinement.

The choices of SMT solver and strongest-postcondition operator will be discussed below.

15.4. CPAchecker Configuration

In order to precisely evaluate the implementation of our framework for predicate analysis in CPAchecker, we have disabled other components of CPAchecker that may be a threat to the validity of the results. Most notably, most configurations of CPAchecker (including those that are part of submissions to SV-COMP) enable so-called counterexample checks (cf. Sect. 5.4.2): A found specification violation is only reported if a second analysis (which can be another configuration of CPAchecker, or the verifier Cbmc) confirms the concrete error path. No such counterexample checks were enabled for the experiments here, which makes the results not directly comparable for example to SV-COMP results of CPAchecker (with counterexample checks, a lower number of wrong and correct alarms is expected).

For optimal performance, we disabled all unnecessary output of CPAchecker (including witness files, a further difference from SV-COMP submissions). We also disabled a heuristic that lets CPAchecker abort immediately in order to avoid wasting time if it encounters a large array such as those in the category *ReachSafety-Arrays*.

In summary, we specified the following options for all runs of CPAchecker:

```
-noout
-disable-java-assertions
-setprop cfa.showDeadCode=false
-setprop analysis.checkCounterexamples=false
-setprop cpa.predicate.abortOnLargeArrays=false
-setprop cpa.predicate.memoryAllocationsAlwaysSucceed=true
```

The last option enables the semantics of `malloc` never returning `null` as assumed by the SV-COMP benchmark suite.

As discussed in Sect. 6.3.1, if using native SMT solvers it is required to reduce the amount of memory available to the JVM in order to reserve memory for the SMT solver. We determined the amount of memory necessary for each native solver depending on the used SMT theories (between 2 GB and 4 GB) and let the JVM use the remainder of the 15 GB of our total memory limit. For Princess and Z3, which use recursive algorithms, we increased the stack size appropriately. These adjustments are necessary for a fair comparison of the solvers.

15.5. Hardware and Software Environment

We specified resource limits that are identical to those of SV-COMP'17: 15 min of CPU time, 15 GB of memory, and 8 processing units of a CPU.

Because our study of the influence of hardware characteristics on the performance of parallel tool executions (cf. Chapter 11) showed that we cannot reliably avoid nondeterministic performance influences if multiple instances of the tool are executed in parallel, we used a dedicated machine for each tool execution. Our benchmarking machines had a single Intel Xeon E3-1230v5 CPU with a frequency of 3.4 GHz and 33 GB of RAM. The CPU has 4 physical cores and hyperthreading (i.e., 8 virtual cores). Turbo Boost, i.e., dynamically increasing the CPU frequency depending on how many cores are used, was disabled. The operating system was Ubuntu 16.04 (64-bit) with Linux 4.4. For CPAchecker we used OpenJDK 1.8 as JVM. This is the same system specification as used in SV-COMP'17.

In order to fulfill all our requirements for reliable benchmarking from Chapter 9 and avoid the problems shown in Chapter 10, we used BenchExec[4] (cf. Chap-

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

ter 13) in version 1.9 as benchmarking framework, i.e., the tool was executed in a benchmarking container built with cgroups and namespaces (cf. Chapter 12).

15.6. Replicability and Availability of Results

Everything that is necessary to replicate our experiments is available online on the supplementary webpage[5]. We provide an archive with the benchmark definitions in the format for BenchExec, which can be used to directly rerun the experiments. For example, these benchmark definitions specify the resource limits, all command-line arguments for each benchmarked configuration, and the set of verification tasks.

CPAchecker in the benchmarked version can be downloaded from the supplementary webpage. The source code of CPAchecker is available under the Apache 2.0 license from the official repository[6]. Archives with the other verifiers that are used for comparison are available via links from the SV-COMP webpage[7]. The input files for the verification tasks are available in the official repository[8].

Tables with the full results are also available on the supplementary webpage. We also provide the raw measurement results in the BenchExec format. The log files with the tool output (`stdout` and `stderr`) of all tool executions are not available online for space reasons (even compressed, these are several gigabytes), but can be requested from the author.

15.7. Presentation of Results

For the presentation of measurement results, we use SI units and prefixes. This means, for example, that 1 MB is exactly 1 000 000 B. We round measurement results to three significant digits.

For the visualization and comparison of CPU-time results of several tool configurations we use quantile plots. In these plots, a data point (x, y) is shown if the respective configuration could solve x verification tasks in at most y seconds. This leads to a plot where each graph represents the performance of one

[5] `https://www.sosy-lab.org/research/phd/wendler`
[6] `https://cpachecker.sosy-lab.org/download.php`
[7] `https://sv-comp.sosy-lab.org/2017/systems.php`
[8] `https://github.com/sosy-lab/sv-benchmarks`, Git tag `svcomp17`

configuration and the graphs are monotonically increasing. Only data points for correct results are shown. Thus, the right-most x-value of each graph in a quantile plot corresponds to the number of correct results that the respective configuration produced. The area under each graph roughly indicates the sum of the run times for all correct results of the configuration. Thus, in general, a configuration could be considered "better" the further to the right its graph stretches and the lower it is. Furthermore, the slope of a graph may indicate how the configuration scales for more difficult tasks.

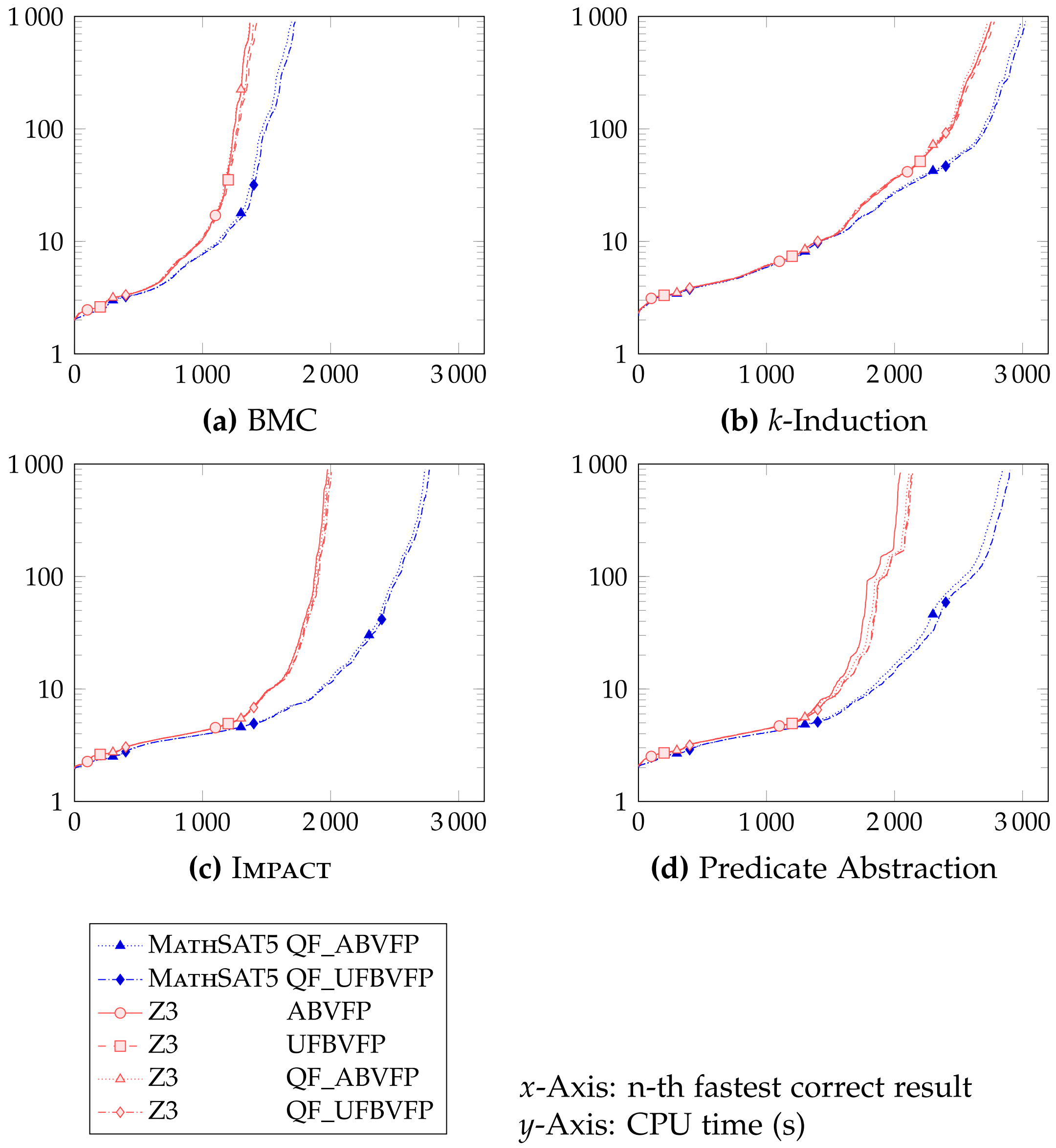
16. Comparison of SMT Solvers and Theories

When evaluating algorithms and approaches for software model checking, the choice of the SMT solver and the theories used to encode program semantics is often neglected, because this choice is conceptually orthogonal to any choices regarding the verification approach (this can for example be seen in Table 5.1). However, it is not guaranteed that this orthogonality actually holds in practice. It could be the case that different SMT solvers and theories influence results of experimental studies of verification approaches and bias results for or against certain approaches, and it is not obvious which combination of SMT solver and theories is most suited for meaningful experiments with verification approaches. Thus, we conduct a study that compares all available combinations of SMT solvers and theories across the four configurations of our unifying framework for predicate analysis from Sect. 15.3, which represent four different verification approaches. Furthermore, we would like to find out which of these combinations is the most effective and efficient one, i.e., solves the most verification tasks in the least time. This includes the question whether it is worth choosing an encoding of program semantics that is unsound (i.e., using linear arithmetic or a bounded heap).

The eight available options for SMT theories in our framework are shown in Table 4.2 and the four SMT solvers that are available in our implementation are shown in Table 6.1. Because not all SMT solvers support all theories, this results in 18 possible combinations for each of the four verification approaches that we use in this study. Comparing 18 different combinations of SMT solvers and theories across four verification approaches results in a large number of results, so we split them into two parts. Furthermore, we show only numbers of correctly and incorrectly solved tasks and CPU-time measurements. More detailed results are available in the full tables online.

Table 16.1.: Results for bitprecise SMT theories across all available SMT solvers and algorithms

			Number of Results				Avg. CPU Time (s)	
			Correct		Wrong		Correct	
			Proofs	Alarms	Proofs	Alarms	Proofs	Alarms
BMC	MathSAT5	QF_ABVFP	736	960	0	20	21.4	81.8
		QF_UFBVFP	736	987	0	23	15.3	78.3
	Z3	ABVFP	697	673	10	12	16.9	64.8
		UFBVFP	702	721	12	12	18.7	69.9
		QF_ABVFP	679	688	0	12	16.3	64.7
		QF_UFBVFP	670	726	0	9	15.8	60.2
k-Induction	MathSAT5	QF_ABVFP	2 263	726	0	12	39.7	87.7
		QF_UFBVFP	2 276	749	0	15	40.0	89.3
	Z3	ABVFP	2 203	553	9	10	51.5	84.3
		UFBVFP	2 193	587	10	7	50.3	90.6
		QF_ABVFP	2 167	561	0	10	50.1	85.2
		QF_UFBVFP	2 168	588	0	5	50.0	90.9
Impact	MathSAT5	QF_ABVFP	1 955	784	1	32	26.1	53.9
		QF_UFBVFP	1 960	812	0	46	28.4	50.5
	Z3	ABVFP	1 523	454	5	30	17.3	59.6
		UFBVFP	1 520	480	5	33	17.8	55.1
		QF_ABVFP	1 516	470	0	30	16.4	55.8
		QF_UFBVFP	1 518	487	0	34	15.4	51.8
Pred. Abs.	MathSAT5	QF_ABVFP	2 096	749	1	34	33.7	71.7
		QF_UFBVFP	2 125	777	0	40	32.1	67.7
	Z3	ABVFP	1 640	406	4	30	23.1	67.7
		UFBVFP	1 703	430	7	33	22.8	65.7
		QF_ABVFP	1 699	417	0	30	23.8	69.1
		QF_UFBVFP	1 703	439	0	33	22.7	66.4



(a) BMC

(b) *k*-Induction

(c) Impact

(d) Predicate Abstraction

x-Axis: n-th fastest correct result
y-Axis: CPU time (s)

Figure 16.1.: Quantile plots for CPU time of bitprecise SMT theories across all available SMT solvers and algorithms

16.1. Bitprecise Theories

First, we look only at configurations using bitprecise SMT theories. A summary of the results can be seen in Table 16.1 and quantile plots with the time results for correct results can be seen in Fig. 16.1.

We can see that there is only a small number of wrong proofs, almost exclusively for configurations using quantifiers. The number of wrong alarms is higher across all SMT theories, which is related to remaining imprecisions of our implementation, but is at most 1.1 % of the 4 150 safe verification tasks. Differences between SMT solvers with regard to the number of wrong alarms occur due to timeouts, with the exception of four verification tasks that show a difference in the semantics of a floating-point operation between MathSAT5 and Z3.

For the number of correct results and the performance we can see that the differences for the various SMT theories supported by each solver are rather small, even though configurations that use arrays for modeling heap memory tend to solve fewer verification tasks than configurations that use uninterpreted functions. That the differences between SMT theories are small is also confirmed by Fig. 16.1, where the graphs for all configurations using the same solver are close to each other. Note that in the SV-COMP benchmark suite there exists a large number of verification tasks (e.g., in the category *DeviceDriversLinux64*) that use pointers and heap memory, but where these features are not directly relevant for the safety property. Thus, we may conclude that in general using more complex solver features such as arrays or quantifiers does not negatively affect performance in such cases. However, we must also notice that the more powerful pointer encoding does not immediately allow us to solve significantly more verification tasks that heavily rely on arrays, e.g., those from the category *Arrays*. More work needs to be done in order to be able to prove complex array properties.

Because of the similar results across the various SMT theories, in the remainder of this subsection we will consider only the configuration QF_UFBVFP. An interesting difference can be seen when comparing the two solvers MathSAT5 and Z3 across the four algorithms in the plots of Fig. 16.1. For all of the compared algorithms, the number of solved tasks is higher with MathSAT5. However, for Impact and predicate abstraction this difference between the solvers is significantly larger (38 % and 34 % more solved tasks with MathSAT5, respectively) than for BMC and *k*-induction (only 9.6 % and 9.8 % more solved tasks with

MathSAT5). This difference is caused mostly by failures of the interpolation engine, which can occur only for Impact and predicate abstraction, but not for BMC and *k*-induction. Analyzing the output of CPAchecker shows that interpolation fails in the configuration for predicate abstraction for 70 tasks if MathSAT5 is used, but for 662 tasks with Z3. For Impact, the difference is even larger with 71 failures if using MathSAT5 and 1 316 failures if using Z3. As we have discussed in Sect. 6.3.1, the interpolation component of Z3 seems to be unmaintained, and reported bugs do not get fixed. Our experimental results show that these issues affect a large number of verification tasks, and MathSAT5 should be preferred over Z3 for verification approaches that rely on interpolation.

Furthermore, even if we use MathSAT5 for interpolation, any comparison between algorithms that rely on interpolation and other algorithms is still potentially skewed by the 70 tasks where the interpolation engine of MathSAT5 fails. Remember that these results are for the SMT theory combination QF_UFBVFP, which is the only one of the bitprecise configurations considered here for which the existence of quantifier-free interpolants is guaranteed and which could thus be supposed to be the "easiest" configuration for an interpolation engine.

On the one hand, these results highlight that the choice of the SMT solver can create a significant bias in an experimental comparison of verification algorithms, specifically if these algorithms differ in their solver workload. For example, from a comparison of *k*-induction vs. predicate abstraction that uses Z3 one could draw the conclusion that predicate abstraction is not competitive, which is not true if MathSAT5 is used. While many researchers treat the choice of the SMT solver as a confounding variable that just needs to be kept constant in order to avoid unwanted influences, this is not always enough for a comprehensive comparison of algorithms. Instead, the SMT solver needs to be carefully chosen or several SMT solvers need to be considered. This shows that the implementation of verification approaches in an existing framework with a mature and flexible implementation like ours, which for example allows easy switching between SMT solvers, can be valuable for producing meaningful experimental results.

On the other hand, if we want to evaluate verification algorithms not conceptually but for their effectiveness when adopted in practice, we can of course conclude that the fact that *k*-induction does not require interpolation (and thus, is immune to existing solver problems in this regard) is currently an advantage, at least for bitprecise verification.

Table 16.2.: Results for SMT theories with linear arithmetic across all available SMT solvers and algorithms

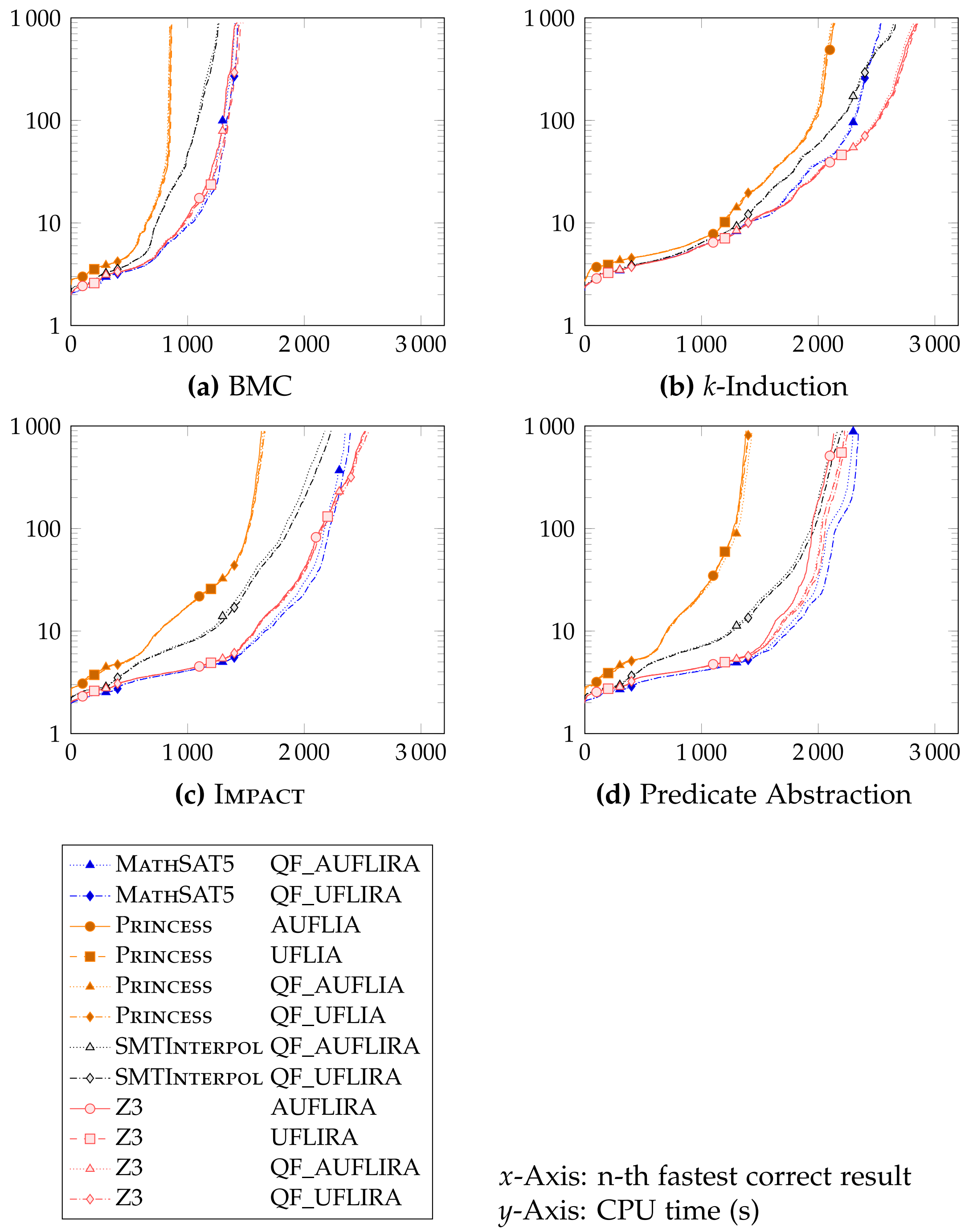
			Number of Results				Avg. CPU Time (s)	
			Correct		Wrong		Correct	
			Proofs	Alarms	Proofs	Alarms	Proofs	Alarms
BMC	MathSAT5	QF_AUFLIRA	682	724	3	98	16.6	42.9
		QF_UFLIRA	683	749	3	102	10.7	41.8
	Princess	AUFLIA	635	223	12	22	13.2	13.7
		UFLIA	634	230	12	37	12.4	13.8
		QF_AUFLIA	618	227	9	22	13.6	16.2
		QF_UFLIA	620	228	9	22	13.2	14.2
	SMTInterpol	QF_AUFLIRA	687	572	2	101	20.6	104
		QF_UFLIRA	691	577	2	104	18.7	102
	Z3	AUFLIRA	703	721	15	55	13.2	77.1
		UFLIRA	718	757	15	57	14.1	70.1
		QF_AUFLIRA	683	742	2	58	12.9	70.4
		QF_UFLIRA	687	769	2	63	13.2	66.3
k-Induction	MathSAT5	QF_AUFLIRA	2 024	513	4	25	38.5	85.9
		QF_UFLIRA	2 018	519	4	29	37.2	79.9
	Princess	AUFLIA	1 967	175	12	5	46.8	20.1
		UFLIA	1 960	180	12	19	43.5	23.8
		QF_AUFLIA	1 941	175	9	5	44.1	20.3
		QF_UFLIA	1 953	177	9	5	46.2	27.1
	SMTInterpol	QF_AUFLIRA	2 192	457	4	24	64.1	161
		QF_UFLIRA	2 203	464	4	27	65.6	161
	Z3	AUFLIRA	2 265	576	16	22	46.9	97.0
		UFLIRA	2 265	598	17	23	47.1	98.7
		QF_AUFLIRA	2 232	589	4	23	46.3	97.9
		QF_UFLIRA	2 244	608	4	26	46.9	94.3

Continued on next page

Table 16.2.: Results for SMT theories with linear arithmetic across all available SMT solvers and algorithms

Continued from previous page

			Number of Results				Avg. CPU Time (s)	
			Correct		Wrong		Correct	
			Proofs	Alarms	Proofs	Alarms	Proofs	Alarms
Impact	MathSAT5	QF_AUFLIRA	1 743	608	4	144	18.9	69.3
		QF_UFLIRA	1 744	653	2	153	20.2	64.0
	Princess	AUFLIA	1 401	233	6	65	36.6	50.9
		UFLIA	1 404	254	6	83	38.3	61.1
		QF_AUFLIA	1 401	243	5	65	34.7	59.9
		QF_UFLIA	1 412	252	5	68	38.5	70.1
	SMTInterpol	QF_AUFLIRA	1 652	525	2	145	51.0	127
		QF_UFLIRA	1 673	558	2	153	48.9	125
	Z3	AUFLIRA	1 760	758	12	101	34.9	118
		UFLIRA	1 762	764	14	103	38.1	111
		QF_AUFLIRA	1 747	780	2	106	37.0	114
		QF_UFLIRA	1 754	799	2	104	39.8	112
Predicate Abstraction	MathSAT5	QF_AUFLIRA	1 743	558	2	136	20.0	58.4
		QF_UFLIRA	1 766	582	2	145	18.3	51.3
	Princess	AUFLIA	1 162	221	5	65	37.9	48.0
		UFLIA	1 160	235	5	82	38.3	51.5
		QF_AUFLIA	1 203	231	5	65	40.0	67.3
		QF_UFLIA	1 168	234	5	65	37.4	57.8
	SMTInterpol	QF_AUFLIRA	1 710	459	2	137	40.8	113
		QF_UFLIRA	1 718	491	2	146	39.6	116
	Z3	AUFLIRA	1 670	463	11	98	26.8	66.8
		UFLIRA	1 736	490	11	102	27.0	67.3
		QF_AUFLIRA	1 740	491	4	104	27.8	69.2
		QF_UFLIRA	1 749	504	4	103	28.9	61.4



(a) BMC

(b) *k*-Induction

(c) Impact

(d) Predicate Abstraction

x-Axis: n-th fastest correct result
y-Axis: CPU time (s)

Figure 16.2.: Quantile plots for CPU time of SMT theories with linear arithmetic across all available SMT solvers and algorithms

16.2. Theories with Linear Arithmetic

Now we consider the remaining SMT theories, i.e., those that provide only linear arithmetic. The summarized results are shown in Table 16.2 and the CPU time for the correct results in Fig. 16.2. Note that because the SMT solver PRINCESS does not support reals, we approximate all program variables with integers for this solver.

We can see that the number of wrong results is higher than if using bitprecise theories, especially the number of wrong alarms. This is expected due to the imprecise approximation of bitvectors. The reason for the wrong proofs are the unsound approximation of floats and the same quantifier-related problem that also occurs for the bitprecise configurations. Note that for PRINCESS more wrong proofs occur because of the approximation of floats with integers instead of rationals. Apart from this, the difference in the number of wrong results between the various solvers can be explained mostly with the different performance of the solvers.

For the bitprecise configurations we noticed that the use of quantifiers and arrays made little difference, and the same is true for the linear configurations.

When comparing the four SMT solvers, we can see that PRINCESS solves significantly fewer tasks than the other solvers. This is due to a larger number of timeouts. The fact that at the same time the average time per solved task for PRINCESS is comparable to those of other solvers indicates that other solvers scale better, i.e., perform better especially for more difficult tasks. This is also observable in the plots of Fig. 16.2, where the graphs for PRINCESS are visibly steeper at their right end than the graphs of the other solvers. The difference between MATHSAT5 and Z3 is much smaller for theories with linear arithmetic than for bitprecise theories, and neither of them consistently outperforms the other. The biggest difference in the results for MATHSAT5 and Z3 occurs for *k*-induction with QF_UFLIRA, where Z3 solves 12 % more tasks. The interpolation problems of Z3 seem to not affect linear theories, because Z3 solves the most tasks of all solvers for IMPACT. SMTINTERPOL is competitive with MATHSAT5 and Z3 for *k*-induction and predicate abstraction, but falls behind for BMC and IMPACT.

Table 16.3.: Results for SMT theories with linear arithmetic (encoding with unsound overflows) across all available SMT solvers and algorithms

			Number of Results				Avg. CPU Time (s)	
			Correct		Wrong		Correct	
			Proofs	Alarms	Proofs	Alarms	Proofs	Alarms
BMC	MathSAT5	QF_AUFLIRA	699	806	7	111	11.5	33.9
		QF_UFLIRA	698	844	7	126	7.42	32.8
	Princess	AUFLIA	732	440	16	39	20.8	59.8
		UFLIA	720	501	16	89	22.2	60.0
		QF_AUFLIA	716	456	13	39	22.0	63.7
		QF_UFLIA	711	486	13	47	19.8	55.4
	SMTInterpol	QF_AUFLIRA	702	980	6	98	11.7	101
		QF_UFLIRA	699	990	6	113	9.06	99.8
	Z3	AUFLIRA	729	759	17	43	12.5	44.7
		UFLIRA	739	794	17	43	12.8	46.0
		QF_AUFLIRA	708	783	6	48	12.4	45.4
		QF_UFLIRA	705	812	6	64	12.8	43.1
k-Induction	MathSAT5	QF_AUFLIRA	2 136	661	22	25	32.8	73.4
		QF_UFLIRA	2 143	693	22	28	32.3	75.2
	Princess	AUFLIA	2 045	285	18	11	44.2	54.0
		UFLIA	2 024	324	18	42	42.7	56.9
		QF_AUFLIA	2 014	295	15	11	44.1	59.9
		QF_UFLIA	2 014	315	15	15	43.7	54.7
	SMTInterpol	QF_AUFLIRA	2 254	683	20	27	38.2	66.4
		QF_UFLIRA	2 256	691	20	31	39.1	64.5
	Z3	AUFLIRA	2 290	620	29	21	38.9	53.6
		UFLIRA	2 299	644	30	21	39.6	55.3
		QF_AUFLIRA	2 254	635	18	21	37.7	51.6
		QF_UFLIRA	2 274	654	19	26	38.9	52.9

Continued on next page

Table 16.3.: Results for SMT theories with linear arithmetic (encoding with unsound overflows) across all available SMT solvers and algorithms

Continued from previous page

			Number of Results				Avg. CPU Time (s)	
			Correct		Wrong		Correct	
			Proofs	Alarms	Proofs	Alarms	Proofs	Alarms
Impact	MathSAT5	QF_AUFLIRA	1 963	792	15	222	16.0	32.4
		QF_UFLIRA	1 968	836	15	238	16.8	27.9
	Princess	AUFLIA	1 706	511	20	78	23.8	74.6
		UFLIA	1 699	582	22	125	23.2	74.6
		QF_AUFLIA	1 699	527	11	78	22.8	80.2
		QF_UFLIA	1 701	559	11	88	23.5	71.6
	SMTInterpol	QF_AUFLIRA	1 907	777	11	148	31.9	82.1
		QF_UFLIRA	1 916	817	11	171	31.4	82.3
	Z3	AUFLIRA	1 952	781	19	94	34.7	47.8
		UFLIRA	1 951	790	22	94	36.2	47.0
		QF_AUFLIRA	1 935	802	10	102	36.1	47.6
		QF_UFLIRA	1 932	820	10	110	34.6	48.6
Predicate Abstraction	MathSAT5	QF_AUFLIRA	2 041	788	11	203	23.6	51.3
		QF_UFLIRA	2 086	836	12	222	22.0	45.7
	Princess	AUFLIA	1 585	443	13	79	25.8	72.3
		UFLIA	1 702	514	13	119	24.6	72.0
		QF_AUFLIA	1 658	453	13	79	28.3	74.5
		QF_UFLIA	1 701	486	13	85	24.5	68.4
	SMTInterpol	QF_AUFLIRA	2 120	747	13	157	46.3	67.7
		QF_UFLIRA	2 135	794	13	178	43.1	66.6
	Z3	AUFLIRA	1 858	598	19	100	23.4	70.8
		UFLIRA	1 934	622	20	104	23.0	71.6
		QF_AUFLIRA	1 949	627	12	114	24.5	67.1
		QF_UFLIRA	1 942	641	13	121	24.0	66.2

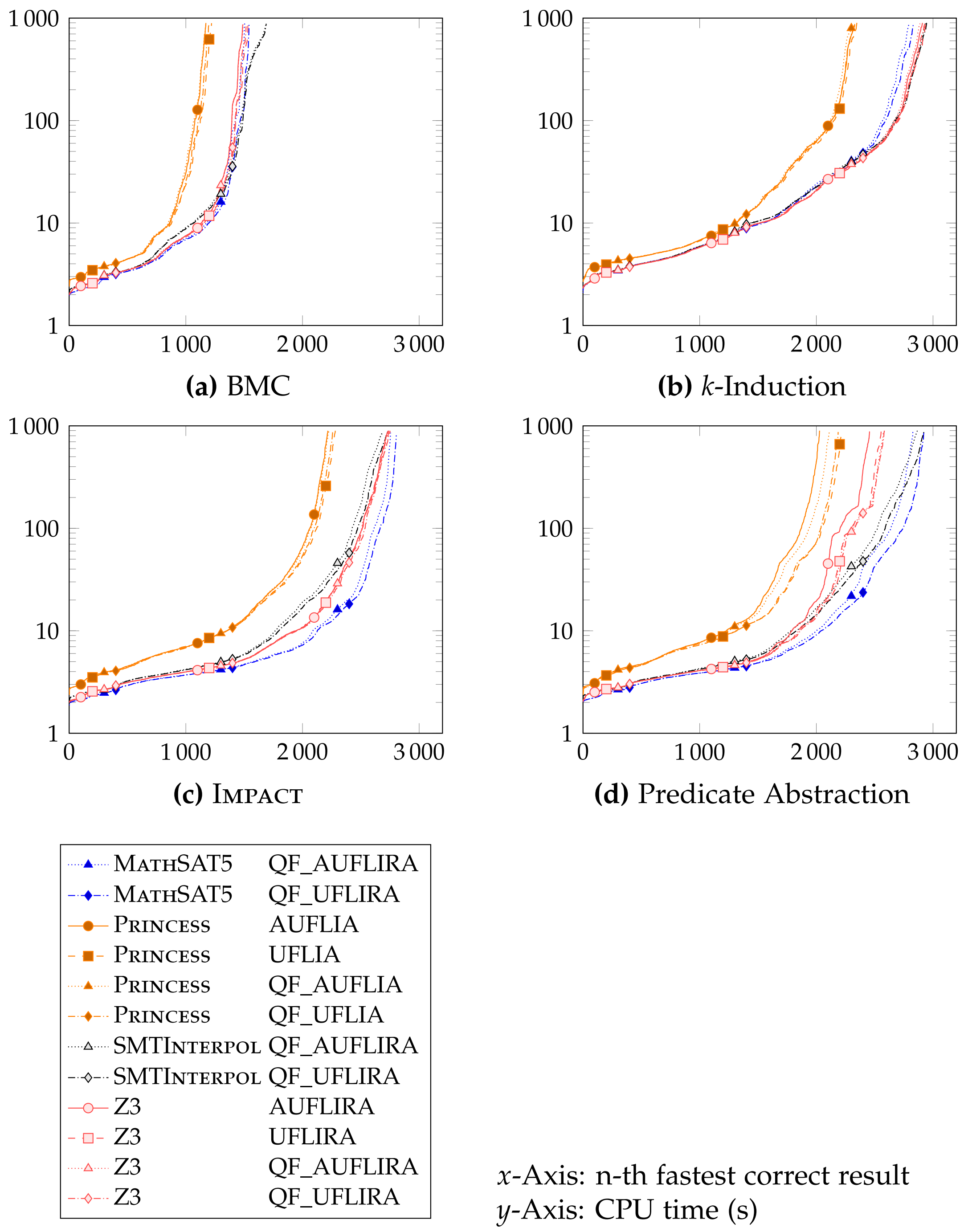


Figure 16.3.: Quantile plots for CPU time of SMT theories with linear arithmetic using unsound overflows across all available SMT solvers and algorithms

16.3. Comparison of Bitprecise and Linear Theories

A comparison of the results for bitprecise theories from Sect. 16.1 and theories with linear arithmetic from Sect. 16.2 shows that for MathSAT5 the number of solved results is sometimes significantly higher for the bitprecise theories. This cannot be fully explained by the change in the number of wrong results. Because it may be unexpected that using bitprecise theories allows solving more tasks, we investigate this further. An additional difference in the encoding of program semantics between using bitprecise theories and theories with linear arithmetic is that for the latter additional disjunctions are needed for every arithmetic operation in order to handle overflows soundly (cf. Sect. 4.4.1). These disjunctions might make solving more difficult. In order to validate this hypothesis, we repeat the experiments from Sect. 16.2 with a changed encoding of program semantics that ignores potential overflows completely (i.e., an arithmetic operator of C is approximated directly with its linear equivalent, without additional disjunctions). The results for these experiments are shown in Table 16.3 and Fig. 16.3. For an easier comparison, we also repeat the results for the theory combinations without arrays and quantifiers (QF_UFBVFP and QF_UFLIRA) from Tables 16.1, 16.2, and 16.3 for all solvers except Princess in Table 16.4 and show plots with the CPU time of these results in Fig. 16.4. Comparing the theory combinations QF_ABVFP and QF_AUFLIRA produces similar results.

The unsound linear encoding indeed seems to be significantly easier for solvers than the sound linear encoding with the many additional disjunctions. In particular, this is the case for SMTInterpol, which is fully competitive with other solvers and even outperforms them in some cases if the unsound linear encoding is used. Z3 performs better with both variants of the linear encoding of program semantics than with the bitprecise encoding, especially for interpolation-based verification approaches due to the problems discussed in Sect. 16.1. MathSAT5, however, correctly solves a similar or even higher number of tasks with the bitprecise encoding than with the linear encoding, even in its unsound variant. The performance of MathSAT5 is highest for the unsound linear encoding, but at the cost of fewer correct results and more wrong results.

Table 16.4.: Comparison of SMT theory combinations QF_UFBVFP and QF_UFLIRA; * marks rows with unsound encoding of overflows.

			Number of Results				Avg. CPU Time (s)	
			Correct		Wrong		Correct	
			Proofs	Alarms	Proofs	Alarms	Proofs	Alarms
BMC	MathSAT5	QF_UFBVFP	736	987	0	23	15.3	78.3
		QF_UFLIRA	683	749	3	102	10.7	41.8
		QF_UFLIRA*	698	844	7	126	7.42	32.8
	SMTInterpol	QF_UFLIRA	691	577	2	104	18.7	102
		QF_UFLIRA*	699	990	6	113	9.06	99.8
	Z3	QF_UFBVFP	670	726	0	9	15.8	60.2
		QF_UFLIRA	687	769	2	63	13.2	66.3
		QF_UFLIRA*	705	812	6	64	12.8	43.1
k-Induction	MathSAT5	QF_UFBVFP	2 276	749	0	15	40.0	89.3
		QF_UFLIRA	2 018	519	4	29	37.2	79.9
		QF_UFLIRA*	2 143	693	22	28	32.3	75.2
	SMTInterpol	QF_UFLIRA	2 203	464	4	27	65.6	161
		QF_UFLIRA*	2 256	691	20	31	39.1	64.5
	Z3	QF_UFBVFP	2 168	588	0	5	50.0	90.9
		QF_UFLIRA	2 244	608	4	26	46.9	94.3
		QF_UFLIRA*	2 274	654	19	26	38.9	52.9
Impact	MathSAT5	QF_UFBVFP	1 960	812	0	46	28.4	50.5
		QF_UFLIRA	1 744	653	2	153	20.2	64.0
		QF_UFLIRA*	1 968	836	15	238	16.8	27.9
	SMTInterpol	QF_UFLIRA	1 673	558	2	153	48.9	125
		QF_UFLIRA*	1 916	817	11	171	31.4	82.3
	Z3	QF_UFBVFP	1 518	487	0	34	15.4	51.8
		QF_UFLIRA	1 754	799	2	104	39.8	112
		QF_UFLIRA*	1 932	820	10	110	34.6	48.6
Predicate Abstraction	MathSAT5	QF_UFBVFP	2 125	777	0	40	32.1	67.7
		QF_UFLIRA	1 766	582	2	145	18.3	51.3
		QF_UFLIRA*	2 086	836	12	222	22.0	45.7
	SMTInterpol	QF_UFLIRA	1 718	491	2	146	39.6	116
		QF_UFLIRA*	2 135	794	13	178	43.1	66.6
	Z3	QF_UFBVFP	1 703	439	0	33	22.7	66.4
		QF_UFLIRA	1 749	504	4	103	28.9	61.4
		QF_UFLIRA*	1 942	641	13	121	24.0	66.2

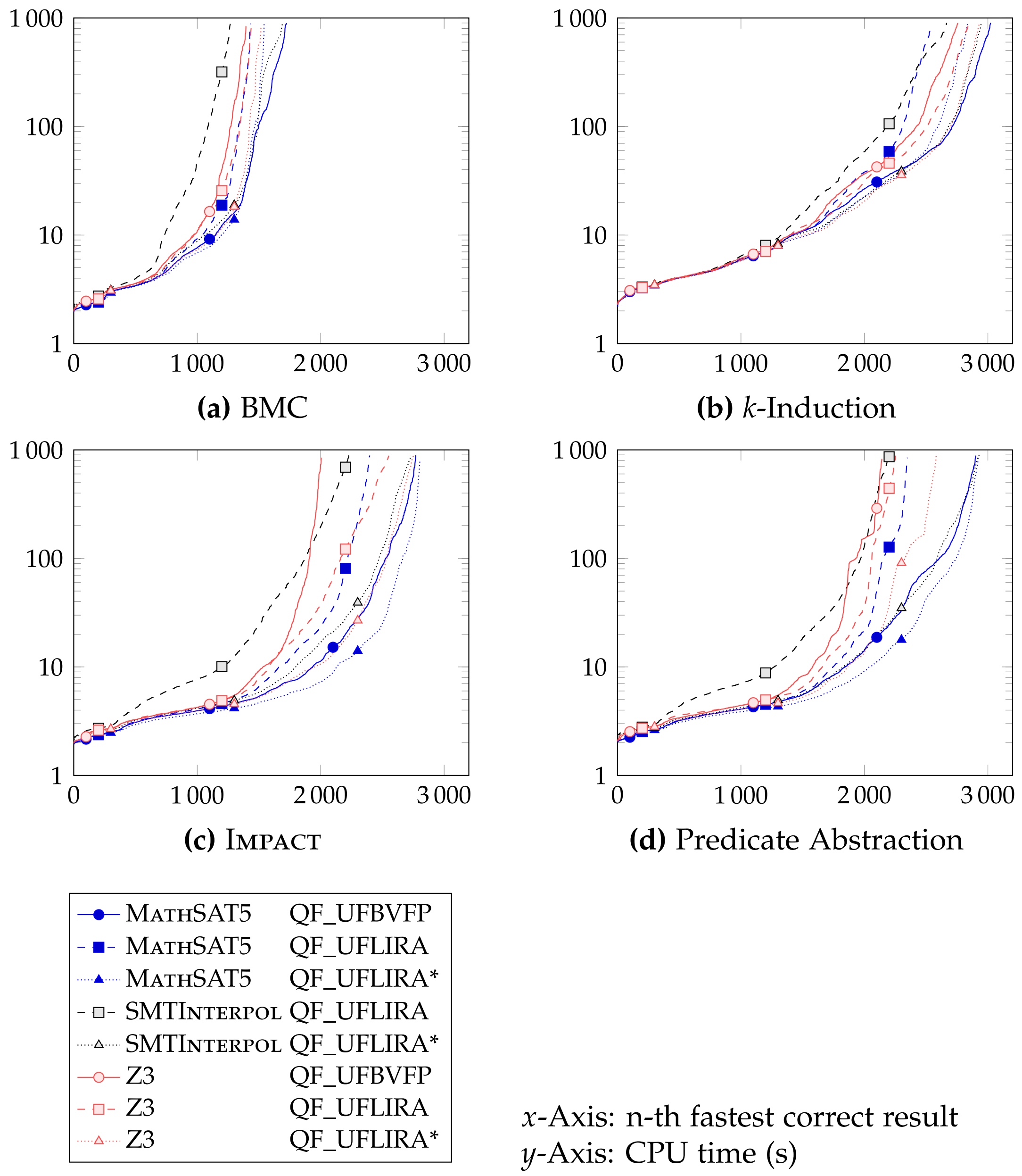


Figure 16.4.: Quantile plots for CPU time of SMT theory combinations QF_UFBVFP and QF_UFLIRA; * marks configurations with unsound encoding of overflows.

16.4. Conclusions

In the following, we summarize the conclusions that we can draw from the results described in this chapter:

- The use of arrays and quantifiers in the formula encoding does not negatively affect performance across all solvers, but also does not help solving more tasks correctly compared to the bounded pointer encoding that uses uninterpreted functions.
- Interpolation for bitprecise theories is still a problem, especially for Z3.
- The choice of the SMT solver does not only significantly affect performance results, but also creates a bias against specific verification approaches, such that drawn conclusions depend on the solver.
- Even small differences in the encoding of program semantics, such as whether overflows are handled for linear arithmetic, can change performance drastically, and create a bias against specific verification approaches.
- The most effective configurations overall are MathSAT5 with one of the bitprecise theory combinations. These configurations produce the least wrong results and the highest or almost the highest number of correct results.

17. Comparison with State of the Art

In order to evaluate the usefulness of the implementation of our unifying framework for predicate analysis, we compare it with other successful software verifiers. The goal of this evaluation is to show that a verifier with a modular and flexible design that supports a high level of configurability is possible without significant overhead. We show two achievements: Our implementation of the verification approaches BMC, *k*-induction, IMPACT, and predicate abstraction is highly competitive in its pure form, and tuned extensions and combinations that are built on top of this implementation are among the best verification tools worldwide.

For the comparison, we take all submissions to SV-COMP'17 [24] that participated in the categories of our benchmark set (cf. Sect. 15.1): 2LS, CBMC, CEAGLE, all CPACHECKER submissions (CPA-BAM-BNB, CPA-KIND, and CPA-SEQ), DEPTHK, all ESBMC submissions (ESBMC, ESBMC-FALSI, ESBMC-INCR, and ESBMC-KIND), SMACK, SYMBIOTIC, and all ULTIMATE submissions (ULTIMATE AUTOMIZER, ULTIMATE KOJAK, and ULTIMATE TAIPAN). This list includes the highest-ranking verifiers of the category *Overall* as well as the highest-ranking verifiers of the categories *ReachSafety* and *SoftwareSystems*, from which our verification tasks are taken. The CPACHECKER submissions are all partially based on our framework for predicate analysis and our implementation, and show what results can be achieved by combining it with other approaches: CPA-BAM-BNB uses predicate abstraction, CPA-KIND uses *k*-induction, and CPA-SEQ uses a sequential combination of several verification approaches, including *k*-induction and predicate abstraction.

We compare these verifiers that represent the state of the art against our implementation of the Predicate CPA in the pure configurations for each of the four verification approaches we have integrated (cf. Sect. 15.3). Following from the results of Chapter 16, we use MATHSAT5 as SMT solver and the theory combination QF_UFBVFP for encoding program semantics.

Table 17.1.: Results for verifiers from SV-COMP'17 (official raw results) and configurations of our framework for predicate-based software verification

	Number of Results				Avg. CPU Time (s)	
	Correct		Wrong		Correct	
	Proofs	Alarms	Proofs	Alarms	Proofs	Alarms
SV-COMP'17						
2LS	1 951	608	9	32	26.5	65.0
CBMC	419	788	1	0	20.6	40.8
CEAGLE	1 609	587	0	1	12.5	13.1
CPA-BAM-BNB	3 022	797	18	53	41.7	89.8
CPA-KIND	2 263	405	0	5	34.4	70.4
CPA-SEQ	2 665	852	0	5	72.8	53.8
DEPTHK	785	698	0	8	112	240
ESBMC	1 164	480	5	32	114	112
ESBMC-FALSI	0	931	0	6	—	58.0
ESBMC-INCR	367	924	0	6	18.0	75.3
ESBMC-KIND	943	802	1	5	13.5	72.8
SMACK	2 362	830	0	1	198	48.3
SYMBIOTIC	1 205	850	3	7	2.86	47.2
ULTIMATE AUTOMIZER	2 495	464	4	7	63.1	129
ULTIMATE KOJAK	1 169	281	0	0	133	64.0
ULTIMATE TAIPAN	2 148	293	4	7	62.8	69.7
Predicate CPA (MathSAT5 QF_UFBVFP)						
BMC	736	987	0	23	15.3	78.3
k-Induction	2 276	749	0	15	40.0	89.3
IMPACT	1 960	812	0	46	28.4	50.5
Predicate Abstraction	2 125	777	0	40	32.1	67.7

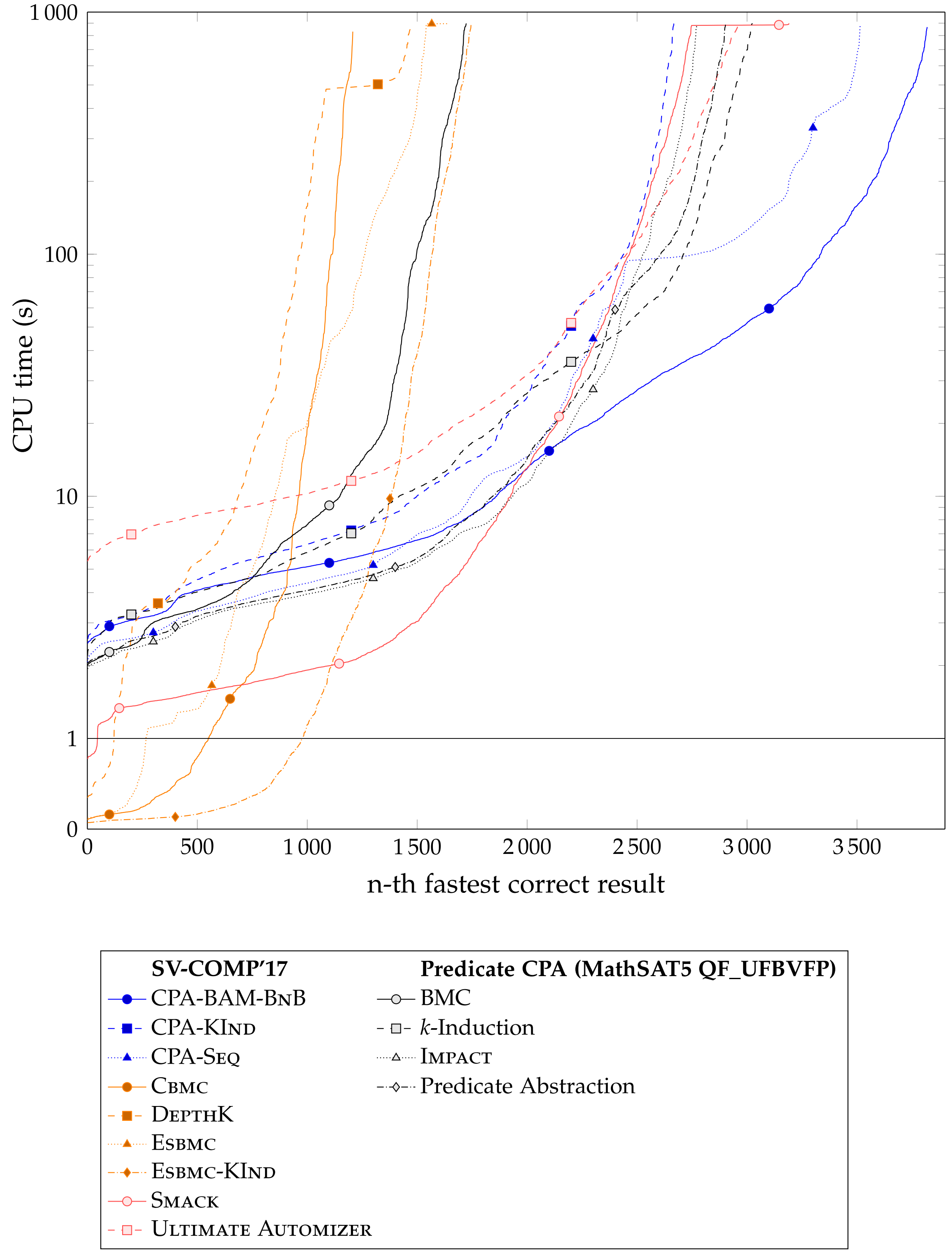


Figure 17.1.: Quantile plots for CPU time of verifiers from SV-COMP'17 (official raw results) and configurations of our framework for predicate-based software verification (linear scale for time range from 0 s to 1 s, logarithmic scale otherwise)

Because our experiment setup is the same as in SV-COMP'17 (cf. Sect. 15.5), we use the official raw results of SV-COMP'17[1]. This ensures the highest possible validity of the results, which have been produced after careful tuning and testing and were approved by the respective tool authors. However, because we are interested in directly comparing the tool effectivity, we use the raw results before witness validation, i.e., each result that a verifier outputs is counted (if it matches the expected verification result). For the SV-COMP'17 ranking, results are only counted fully if the witness that was produced by the verifier can be validated independently. Note that our presented summary results further differ from the official SV-COMP results because we use only a subset of the verification tasks (cf. Sect. 15.1) and we do not use the scoring and weighting scheme of SV-COMP'17.

The results for this comparison are shown in Table 17.1. Figure 17.1 contains a quantile plot with the CPU time for correct results of the configurations of our unifying framework as well as for the most relevant verifiers of SV-COMP'17. In order to clutter the plot not too much we show only the CPAchecker configurations as well as Cbmc and Esbmc because they are similar to our BMC configuration, DepthK and Esbmc-KInd because they are similar to our *k*-induction configuration, and Smack and Ultimate Automizer because they have a competitive number of correctly solved tasks.

Most SV-COMP'17 verifiers in our comparison achieve a lower number of wrong alarms than our Predicate CPA. However, note that several of them use further tools to double check found specification violations in order to reduce the number of wrong alarms, which is possible with our Predicate CPA as well; we explicitly disabled such counterexample checks to avoid unwanted influences on the results (cf. Sect. 15.4). CPA-BAM-BnB and CPA-Seq produce significantly more correct proofs than the four configurations of our unifying framework: both verifiers are partially based on our implementation and benefit from combining it with other verification approaches that have complementary strengths. Smack has more correct proofs than any pure configuration of our Predicate CPA, but we can see in Fig. 17.1 from the horizontal line at the right end of Smack's graph that in many cases Smack terminates right before the timeout and outputs a guessed result. This is also the reason for the high average CPU time of

[1] Available on `https://sv-comp.sosy-lab.org/2017/results/results-verified/`

Smack. From the plot we can see that without these guesses, Smack would produce fewer correct results than the configurations for *k*-induction, Impact, and predicate abstraction of our framework for predicate analysis. Ultimate Automizer and Ultimate Taipan also produce a high number of correct proofs, but both have 4 wrong proofs and a low number of correct alarms. All other SV-COMP'17 verifiers have fewer correct proofs and fewer correct results overall for our benchmark set than our unifying framework for predicate analysis in its unbounded configurations. Furthermore, the BMC configuration of our framework finds more correct alarms in our benchmark set than any of the SV-COMP'17 verifiers. The average CPU time per correctly solved verification tasks for the configurations of our unifying framework is in the range of the average CPU time of the SV-COMP'17 verifiers, for both correct proofs and correct alarms, and even if we exclude the disproportionately high CPU time of Smack.

In summary, this comparison shows that our implementation is highly competitive with the best verifiers of SV-COMP'17 in both the number of correct results as well as performance.

18. Summary and Future Work

This concluding chapter summarizes the work presented in this thesis and lists possible extensions in the future.

18.1. Summary

In Chapter 4 we have developed a unifying framework for software verification based on predicates. Our framework is highly flexible and configurable, and provides many possible choices, allowing us to express the existing and successful software-verification approaches bounded model checking (BMC) [54], *k*-induction [217], IMPACT [190], and lazy predicate abstraction [125, 140] as specific instantiations of our framework (cf. Chapter 5). This unification of approaches in a common framework enables easy studying, experimentation with, extending and combining the integrated approaches, preparing the grounds for all kinds of further research in the domain of software verification using predicates. The amount of research projects that have already benefited from this work [28, 29, 30, 31, 36, 43, 45, 47, 52, 75, 164, 233] shows that our framework is already successful in enabling thriving research. Furthermore, our study in Chapter 16 on how the choice of SMT solvers and SMT theories affect verification approaches, in which 120 different configurations were evaluated, would not have been possible without such a flexible and configurable unifying framework.

The open-source implementation of our framework for predicate analysis in CPACHECKER, which is described in Chapter 6, supports the verification of sequential C programs. This implementation is used in practice for verification of Linux kernel drivers [165], where it has found several dozens of real kernel bugs, and by the International Competition on Software Verification (SV-COMP) as one of two witness validators for validating the results of all tools [23, 24]. The predicate analysis of CPACHECKER has won four medals, including a gold medal, in its first year of participation in SV-COMP [19], and helped to win another 36 medals in six years of SV-COMP as a part of other submissions (cf. Sect. 1.1.2 for full list). These achievements of CPAchecker in SV-COMP have also been awarded by the Kurt Gödel Society with a Gödel medal at the Vienna Summer of Logic 2014.[1] Our evaluation in Chapter 17 shows that our implementation is highly competitive and can outperform many existing software verifiers.

We have investigated in Chapters 9, 10, and 11 what needs to be fulfilled for reliable benchmarking of verifiers. The requirements that we established can serve as a checklist for experimenters as well as reviewers of reports on experimental results. Because we have found that existing benchmarking solutions can cause arbitrarily large measurement errors in practice, we developed our open-source and easy-to-use benchmarking framework BenchExec, which is described in Chapter 13. It builds benchmarking containers using the Linux kernel techniques cgroups and namespaces (cf. Chapter 12) in order to ensure that all our requirements for reliable benchmarking are fulfilled, e.g., a proper isolation of runs.

BenchExec has not only made it possible to perform an experimental study with 671 280 verification runs and 3 620 days of CPU time for this thesis, but is already used by several research groups in their work on developing verification tools. Furthermore, it is the technical foundation of SV-COMP [23], where (in 2017) 32 different submissions were benchmarked, with all results approved by the respective tool authors. The experience of the SV-COMP organization shows that, without the techniques on which BenchExec is based, large measurement errors would have occurred in past instances of SV-COMP.

A preliminary version of our framework for predicate analysis was published at FMCAD'12 [52], and an updated version is to appear in the Springer journal JAR [33]. Its implementation was described in a series of reports for submissions to SV-COMP [100, 186, 187, 231]. A preliminary version of our requirements for reliable benchmarking and BenchExec was published at SPIN'15 [46], and an updated version was accepted with minor revisions by the Springer journal STTT [49].

In summary, we have provided a unifying framework for software verification based on predicates, with a mature and efficient implementation, as an alternative

[1] `https://cpachecker.sosy-lab.org/achieve.php#awards`

to the existing fragmented verification algorithms and tools. We established a solution for reliable benchmarking that makes it possible to effectively evaluate verification approaches. This brings research on predicate-based software verification and its usage in practice onto the next level towards practical predicate analysis.

18.2. Future Work

In the following, we provide a few interesting directions for possible future work.

18.2.1. Predicate Analysis

The most interesting extension of our framework for predicate analysis would be to integrate more algorithms, in particular some variants of IC3/PDR [62, 113]. One possibility would be to implement the clause-learning strategy of PDR as a source of abstract facts for refinement [78]. Integrating the approach of MCMT [122, 123] into our framework would allow comparing MCMT (which can be seen as a version of Impact with backwards reachability) with Impact and other algorithms that are based on forwards reachability.

A goal of MCMT is to prove universally quantified properties over arrays, and a better handling of such properties is interesting also for the existing CEGAR-based approaches, for which techniques are necessary that provide suitable abstract facts about array properties during refinement. In general, for predicate abstraction and Impact the quality of the interpolants that are used during refinement can be crucial for the effectiveness and efficiency of the analysis. Thus, we would like to benefit from some of the techniques that attempt to generate interpolants that are better suited for verification than the interpolants that are returned by off-the-shelf SMT solvers [47, 182].

As we have seen in our experimental evaluation, the choice of the SMT solver and the way how program semantics are encoded into formulas can have a significant impact on the performance of the analysis. Thus, if further interpolating SMT solvers are developed, we plan to integrate them as well. We would also like to further optimize our formula encoding, for example by exploiting knowledge about program variables that can be gained from inferring fine-grained types [8, 105, 213, 225] and bounds [210] for an improved encoding, e.g., with fewer bits for variables with a small range of values [148], or by automatically choosing between a bitprecise encoding and a linear approximation depending on the used operators [96].

18.2.2. Benchmarking

For our benchmarking framework BenchExec, we are interested in extending its measurement capabilities from time and memory to other resources such as energy consumption, either by accessing external power meters or by using the built-in energy-measurement capabilities of modern CPUs [230]. This would allow comparing verification approaches and tools for their energy- and cost-efficiency. A preliminary version of energy measurements was already implemented and used for SV-COMP'17 [24].

Currently the benchmarking containers of BenchExec do not provide a way to limit and measure I/O operations and bandwidth. This is not a problem for benchmarking most verifiers, however, there do exist verifiers that produce a lot of I/O during verification [2], and the ability to limit I/O would improve the repeatability of experiments with these tools.

Furthermore, we are interested in further reducing the mutual performance influences of several parallel runs on a single machine. Making it possible to execute runs for performance experiments in parallel on a single CPU could significantly reduce the necessary wall time for benchmarking in typical verification scenarios, where we have multi-core CPUs but most cores currently remain unused because many verifiers are single-threaded. This would need the ability to control cache allocation of CPUs and limit memory-bandwidth usage.

[2] In SV-COMP'17, one verifier created several GB of temporary files for some verification tasks.

Bibliography

[1] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Craig interpretation. In *Proc. SAS*, LNCS 7460, pages 300–316. Springer, 2012.

[2] A. Albarghouthi, A. Gurfinkel, and M. Chechik. From under-approximations to over-approximations and back. In *Proc. TACAS*, LNCS 7214, pages 157–172. Springer, 2012.

[3] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *Proc. VMCAI*, LNCS 7148, pages 39–55. Springer, 2012.

[4] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. UFO: A framework for abstraction- and interpolation-based software verification. In *Proc. CAV*, LNCS 7358, pages 672–678. Springer, 2012.

[5] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. An extension of lazy abstraction with interpolation for programs with arrays. *Formal Methods in System Design*, 45(1):63–109, 2014.

[6] J. Alglave, A. F. Donaldson, D. Kroening, and M. Tautschnig. Making software verification tools really work. In *Proc. ATVA*, LNCS 6996, pages 28–42. Springer, 2011.

[7] P. Andrianov, K. Friedberger, M. U. Mandrykin, V. S. Mutilin, and A. Volkov. CPA-BAM-BnB: Block-abstraction memoization and region-based memory models for predicate abstractions (competition contribution). In *Proc. TACAS*, LNCS 10206, pages 355–359. Springer, 2017.

[8] S. Apel, D. Beyer, K. Friedberger, F. Raimondi, and A. v. Rhein. Domain types: Abstract-domain selection based on variable usage. In *Proc. HVC*, LNCS 8244, pages 262–278. Springer, 2013.

[9] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *STTT*, 11(1):69–83, 2009.

[10] T. Ball, E. Bounimova, R. Kumar, and V. Levin. SLAM2: Static driver verification with under 4% false alarms. In *Proc. FMCAD*, pages 35–42. IEEE, 2010.

[11] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Proc. IFM*, LNCS 2999, pages 1–20. Springer, 2004.

[12] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *Proc. TACAS*, LNCS 2031, pages 268–283. Springer, 2001.

[13] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.

[14] S. Bardin, A. Finkel, J. Leroux, and P. Schnoebelen. Flat acceleration in symbolic model checking. In *Proc. ATVA*, LNCS 3707, pages 474–488. Springer, 2005.

[15] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *Proc. ACM*, pages 549–560. ACM, 2013.

[16] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, University of Iowa, 2015. Available at www.smt-lib.org.

[17] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In *Proc. SMT*, 2010.

[18] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Proc. DAC*, pages 522–527. ACM, 1998.

[19] D. Beyer. Competition on software verification (SV-COMP). In *Proc. TACAS*, LNCS 7214, pages 504–524. Springer, 2012.

[20] D. Beyer. Second competition on software verification (Summary of SV-COMP 2013). In *Proc. TACAS*, LNCS 7795, pages 594–609. Springer, 2013.

[21] D. Beyer. Status report on software verification (Competition summary SV-COMP 2014). In *Proc. TACAS*, LNCS 8413, pages 373–388. Springer, 2014.

[22] D. Beyer. Software verification and verifiable witnesses (Report on SV-COMP 2015). In *Proc. TACAS*, LNCS 9035, pages 401–416. Springer, 2015.

[23] D. Beyer. Reliable and reproducible competition results with BenchExec and witnesses (Report on SV-COMP 2016). In *Proc. TACAS*, LNCS 9636, pages 887–904. Springer, 2016.

[24] D. Beyer. Software verification with validation of results (Report on SV-COMP 2017). In *Proc. TACAS*, LNCS 10206, pages 331–349. Springer, 2017.

[25] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proc. ICSE*, pages 326–335. IEEE, 2004.

[26] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The Blast query language for software verification. In *Proc. SAS*, LNCS 3148, pages 2–18. Springer, 2004.

[27] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Proc. FMCAD*, pages 25–32. IEEE, 2009.

[28] D. Beyer and M. Dangl. SMT-based software model checking: An experimental comparison of four algorithms. In *Proc. VSTTE*, LNCS 9971, pages 181–198. Springer, 2016.

[29] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. Correctness witnesses: Exchanging verification results between verifiers. In *Proc. FSE*, pages 326–337. ACM, 2016.

[30] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness validation and stepwise testification across software verifiers. In *Proc. FSE*, pages 721–733. ACM, 2015.

[31] D. Beyer, M. Dangl, and P. Wendler. Boosting k-induction with continuously-refined invariants. In *Proc. CAV*, LNCS 9206, pages 622–640. Springer, 2015.

[32] D. Beyer, M. Dangl, and P. Wendler. Combining k-induction with continuously-refined invariants. Technical Report MIP-1503, University of Passau, January 2015. arXiv:1502.00096.

[33] D. Beyer, M. Dangl, and P. Wendler. A unifying view on SMT-based software verification. *J. Autom. Reasoning*, 2017. Accepted, currently under revision.

[34] D. Beyer, G. Dresler, and P. Wendler. Software verification in the Google App-Engine cloud. In *Proc. CAV*, LNCS 8559, pages 327–333. Springer, 2014.

[35] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST. *Int. J. Softw. Tools Technol. Transfer*, 9(5-6):505–525, 2007.

[36] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. Conditional model checking: A technique to pass information between verifiers. In *Proc. FSE*, pages 57:1–57:11. ACM, 2012.

[37] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *Proc. VMCAI*, LNCS 4349, pages 378–394. Springer, 2007.

[38] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *Proc. PLDI*, pages 300–309. ACM, 2007.

[39] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proc. CAV*, LNCS 4590, pages 504–518. Springer, 2007.

[40] D. Beyer, T. A. Henzinger, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *Proc. ASE*, pages 29–38. IEEE, 2008.

[41] D. Beyer and M. E. Keremoglu. CPACHECKER: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.

[42] D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.

[43] D. Beyer and T. Lemberger. Symbolic execution with CEGAR. In *Proc. ISoLA*, LNCS 9952, pages 195–211. Springer, 2016.

[44] D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Proc. FASE*, LNCS 7793, pages 146–162. Springer, 2013.

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

[46] D. Beyer, S. Löwe, and P. Wendler. Benchmarking and resource measurement. In *Proc. SPIN*, LNCS 9232, pages 160–178. Springer, 2015.

[47] D. Beyer, S. Löwe, and P. Wendler. Refinement selection. In *Proc. SPIN*, LNCS 9232, pages 20–38. Springer, 2015.

[48] D. Beyer, S. Löwe, and P. Wendler. Sliced path prefixes: An effective method to enable refinement selection. In *Proc. FORTE*, LNCS 9039, pages 228–243. Springer, 2015.

[49] D. Beyer, S. Löwe, and P. Wendler. Reliable benchmarking: Requirements and solutions. *Int. J. Softw. Tools Technol. Transfer*, 2017. Accepted, currently under revision.

[50] D. Beyer and A. K. Petrenko. Linux driver verification. In *Proc. ISoLA*, LNCS 7610, pages 1–6. Springer, 2012.

[51] D. Beyer and A. Stahlbauer. BDD-based software model checking with CPAchecker. In *Proc. MEMICS*, LNCS 7721, pages 1–11. Springer, 2013.

[52] D. Beyer and P. Wendler. Algorithms for software model checking: Predicate abstraction vs. Impact. In *Proc. FMCAD*, pages 106–113. FMCAD, 2012.

[53] D. Beyer and P. Wendler. Reuse of verification results: Conditional model checking, precision reuse, and verification witnesses. In *Proc. SPIN*, LNCS 7976, pages 1–17. Springer, 2013.

[54] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS*, LNCS 1579, pages 193–207. Springer, 1999.

[55] J. Birgmeier, A. R. Bradley, and G. Weissenbacher. Counterexample to induction-guided abstraction-refinement (CTIGAR). In *Proc. CAV*, LNCS 8559, pages 831–848. Springer, 2014.

[56] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. PLDI*, pages 196–207. ACM, 2003.

[57] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Faculté des Sciences Appliquées de Université de Liège, 1998.

[58] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer, 1997.

[59] R. Bornat. Proving pointer programs in hoare logic. In *Proc. Mathematics of Program Construction*, LNCS 1837, pages 102–126. Springer, 2000.

[60] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *Softw. Test., Verif. Reliab.*, 16(2):97–121, 2006.

[61] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *Proc. CAV*, LNCS 1855, pages 403–418. Springer, 2000.

[62] A. R. Bradley. SAT-based model checking without unrolling. In *Proc. VMCAI*, LNCS 6538, pages 70–87. Springer, 2011.

[63] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Proc. VMCAI*, LNCS 3855, pages 427–442. Springer, 2006.

[64] M. Brain, S. Joshi, D. Kröning, and P. Schrammel. Safety verification and refutation by k-invariants and k-induction. In *Proc. SAS*, LNCS 9291, pages 145–161. Springer, 2015.

[65] G. Brat, J. A. Navas, N. Shi, and A. Venet. Ikos A framework for static analysis based on abstract interpretation. In *Proc. SEFM*, LNCS 8702, pages 271–277. Springer, 2014.

[66] A. Brillout, D. Kroening, and T. Wahl. Mixed abstractions for floating-point arithmetic. In *Proc. FMCAD*, pages 69–76. IEEE, 2009.

[67] A. Brooks, M. Roper, M. Wood, J. Daly, and J. Miller. Replication's role in software engineering. In *Guide to Advanced Empirical Software Engineering*, pages 365–379. Springer, 2008.

[68] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing abstractions. *Fundam. Inform.*, 89(4):369–392, 2008.

[69] R. Bruttomesso, S. Ghilardi, and S. Ranise. Rewriting-based quantifier-free interpolation for a theory of arrays. In *Proc. Int. Conf. on Rewriting Techniques and Applications*, pages 171–186. Schloss Dagstuhl, 2011.

[70] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[71] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[72] N. Caniart, E. Fleury, J. Leroux, and M. Zeitoun. Accelerating interpolation-based model-checking. In *Proc. TACAS*, LNCS 4963, pages 428–442. Springer, 2008.

[73] M. Carter, S. He, J. Whitaker, Z. Rakamaric, and M. Emmi. SMACK software verification toolchain. In *Proc. ICSE (Companion Volume)*, pages 589–592. ACM, 2016.

[74] G. Charwat, G. Ianni, T. Krennwallner, M. Kronegger, A. Pfandler, C. Redl, M. Schwengerer, L. Spendier, J. Wallner, and G. Xiao. VCWC: A versioning competition workflow compiler. In *Proc. LPNMR*, LNCS 8148, pages 233–238. Springer, 2013.

[75] Y. Chen, C. Hsieh, M. Tsai, B. Wang, and F. Wang. Verifying recursive programs using intraprocedural analyzers. In *Proc. SAS*, LNCS 8723, pages 118–133. Springer, 2014.

[76] Y. Chen, C. Hsieh, M. Tsai, B. Wang, and F. Wang. CPArec: Verifying recursive programs via source-to-source program transformation (competition contribution). In *Proc. TACAS*, LNCS 9035, pages 426–428. Springer, 2015.

[77] J. Christ, J. Hoenicke, and A. Nutz. SMTINTERPOL: An interpolating SMT solver. In *Proc. SPIN*, LNCS 7385, pages 248–254. Springer, 2012.

[78] A. Cimatti and A. Griggio. Software model checking via IC3. In *Proc. CAV*, LNCS 7358, pages 277–293. Springer, 2012.

[79] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri. KRATOS: A software model checker for SYSTEMC. In *Proc. CAV*, LNCS 6806, pages 310–316. Springer, 2011.

[80] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. IC3 modulo theories via implicit predicate abstraction. In *Proc. TACAS*, LNCS 8413, pages 46–61. Springer, 2014.

[81] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In *Proc. TACAS*, LNCS 7795, pages 93–107. Springer, 2013.

[82] A. Cimatti, A. Griggio, and R. Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *J. Artif. Intell. Res. (JAIR)*, 40:701–728, 2011.

[83] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proc. Logic of Programs 1981*, LNCS 131, pages 52–71. Springer, 1982.

[84] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[85] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT, 1999.

[86] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. TACAS*, LNCS 2988, pages 168–176. Springer, 2004.

[87] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SatAbs: SAT-based predicate abstraction for ANSI-C. In *Proc. TACAS*, LNCS 3440, pages 570–574. Springer, 2005.

[88] E. M. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. Technical Report CMU-CS-03-126, Carnegie Mellon University, School of Computer Science, 2003.

[89] E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A precise yet efficient memory model for C. *Electr. Notes Theor. Comput. Sci.*, 254:85–103, 2009.

[90] D. R. Cok, D. Déharbe, and T. Weber. The 2014 SMT competition. *JSAT*, 9:207–242, 2016.

[91] C. S. Collberg and T. A. Proebsting. Repeatability in computer-systems research. *Commun. ACM*, 59(3):62–69, 2016.

[92] P. Collingbourne, C. Cadar, and P. H. J. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *Proc. EuroSys*, pages 315–328. ACM, 2011.

[93] M. Colón, S. Sankaranarayanan, and H. B. Sipma. Linear invariant generation using non-linear constraint solving. In *Proc. CAV*, LNCS 2725, pages 420–432. Springer, 2003.

[94] B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In *Proc. CAV*, LNCS 3576, pages 296–300. Springer, 2005.

[95] L. C. Cordeiro, B. Fischer, and J. Marques-Silva. Continuous verification of large embedded software using smt-based bounded model checking. In *Proc. ECBS*, pages 160–169. IEEE Computer Society, 2010.

[96] L. C. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.*, 38(4):957–974, 2012.

[97] W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *J. Symb. Log.*, 22(3):250–268, 1957.

[98] D. Cyrluk, M. O. Möller, and H. Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *Proc. CAV*, LNCS 1254, pages 60–71. Springer, 1997.

[99] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[100] M. Dangl, S. Löwe, and P. Wendler. CPAchecker with support for recursive programs and floating-point arithmetic (competition contribution). In *Proc. TACAS*, LNCS 9035, pages 423–425. Springer, 2015.

[101] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, LNCS 4963, pages 337–340. Springer, 2008.

[102] A. B. de Oliveira, J.-C. Petkovich, and S. Fischmeister. How much does memory layout impact performance? A wide study. In *Proc. REPRODUCE*, 2014.

[103] R. DeLine and R. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[104] Y. Demyanova, P. Rümmer, and F. Zuleger. Systematic predicate abstraction using variable roles. In *Proc. NFM*, LNCS 10227, pages 265–281. Springer, 2017.

[105] Y. Demyanova, H. Veith, and F. Zuleger. On the concept of variable roles and its use in software analysis. In *Proc. FMCAD*, pages 226–230. IEEE, 2013.

[106] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[107] I. Dillig, T. Dillig, B. Li, and K. L. McMillan. Inductive invariant generation via abductive inference. In *Proc. OOPSLA*, pages 443–456. ACM, 2013.

[108] M. Dittrich. Bit-precise predicate analysis with CPAchecker. Bachelor's thesis, University of Passau, Software Systems Lab, 2013.

[109] A. F. Donaldson, L. Haller, and D. Kroening. Strengthening induction-based race checking with lightweight static analysis. In *Proc. VMCAI*, LNCS 6538, pages 169–183. Springer, 2011.

[110] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer. Software verification using k-induction. In *Proc. SAS*, LNCS 6887, pages 351–368. Springer, 2011.

[111] A. F. Donaldson, D. Kroening, and P. Rümmer. Automatic analysis of DMA races using model checking and *k*-induction. *FMSD*, 39(1):83–113, 2011.

[112] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant strength. In *Proc. VMCAI*, LNCS 5944, pages 129–145. Springer, 2010.

[113] N. Eén, A. Mishchenko, and R. K. Brayton. Efficient implementation of property directed reachability. In *Proc. FMCAD*, pages 125–134. FMCAD Inc., 2011.

[114] E. Ermis, J. Hoenicke, and A. Podelski. Splitting via interpolants. In *Proc. VMCAI*, LNCS 7148, pages 186–201. Springer, 2012.

[115] E. Ermis, A. Nutz, D. Dietsch, J. Hoenicke, and A. Podelski. Ultimate Kojak (competition contribution). In *Proc. TACAS*, LNCS 8413, pages 421–423. Springer, 2014.

[116] S. Falke, F. Merz, and C. Sinz. The bounded model checker Llbmc. In *Proc. ASE*, pages 706–709. IEEE, 2013.

[117] R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, pages 19–32. AMS, 1967.

[118] K. Friedberger. Block-abstraction memoization as an approach to verify recursive procedures. Master's thesis, University of Passau, Software Systems Lab, 2015.

[119] K. Friedberger. CPA-BAM: Block-abstraction memoization with value analysis and predicate analysis (competition contribution). In *Proc. TACAS*, LNCS 9636, pages 912–915. Springer, 2016.

[120] M. Y. R. Gadelha, H. I. Ismail, and L. C. Cordeiro. Handling loops in bounded model checking of C programs via k-induction. *STTT*, pages 1–18, 2015.

[121] A. Galloway, G. Lüttgen, J. T. Mühlberg, and R. Siminiceanu. Model-checking the Linux virtual file system. In *Proc. VMCAI*, LNCS 5403, pages 74–88. Springer, 2009.

[122] S. Ghilardi and S. Ranise. Goal-directed invariant synthesis for model checking modulo theories. In *Proc. TABLEAUX*, LNCS 5607, pages 173–188. Springer, 2009.

[123] S. Ghilardi and S. Ranise. Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. *Logical Methods in Computer Science*, 6(4), 2010.

[124] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[125] S. Graf and H. Saïdi. Construction of abstract state graphs with Pvs. In *Proc. CAV*, LNCS 1254, pages 72–83. Springer, 1997.

[126] M. Greitschus, D. Dietsch, M. Heizmann, A. Nutz, C. Schätzle, C. Schilling, F. Schüssele, and A. Podelski. Ultimate Taipan: Trace abstraction and abstract interpretation (competition contribution). In *Proc. TACAS*, LNCS 10206, pages 399–403. Springer, 2017.

[127] M. Greitschus, D. Dietsch, and A. Podelski. Refining trace abstraction using abstract interpretation. *CoRR*, abs/1702.02369, 2017.

[128] A. Griggio. Effective word-level interpolation for software verification. In *Proc. FMCAD*, pages 28–36. FMCAD Inc., 2011.

[129] A. Griggio and M. Roveri. Comparing different variants of the IC3 algorithm for hardware model checking. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 35(6):1026–1039, 2016.

[130] D. Gu, C. Verbrugge, and E. Gagnon. Code layout as a source of noise in JVM performance. *Stud. Inform. Univ.*, 4(1):83–99, 2005.

[131] A. Gurfinkel, A. Albarghouthi, S. Chaki, Y. Li, and M. Chechik. Ufo: Verification with interpolants and abstract interpretation (competition contribution). In *Proc. TACAS*, LNCS 7795, pages 637–640. Springer, 2013.

[132] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In *Proc. CAV*, LNCS 9206, pages 343–361. Springer, 2015.

[133] Á. Hajdu, T. Tóth, A. Vörös, and I. Majzik. A configurable CEGAR framework with interpolation-based refinements. In *Proc. FORTE*, LNCS 9688, pages 158–174. Springer, 2016.

[134] L. Haller, A. Griggio, M. Brain, and D. Kroening. Deciding floating-point logic with systematic abstraction. In *Proc. FMCAD*, pages 131–140. IEEE, 2012.

[135] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proc. CoNEXT*, pages 253–264. ACM, 2012.

[136] M. Heizmann, Y. Chen, D. Dietsch, M. Greitschus, A. Nutz, B. Musa, C. Schätzle, C. Schilling, F. Schüssele, and A. Podelski. Ultimate Automizer with an on-demand construction of floyd-hoare automata (competition contribution). In *Proc. TACAS*, LNCS 10206, pages 394–398. Springer, 2017.

[137] M. Heizmann, J. Hoenicke, and A. Podelski. Refinement of trace abstraction. In *Proc. SAS*, LNCS 5673, pages 69–85. Springer, 2009.

[138] M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *Proc. POPL*, pages 471–482. ACM, 2010.

[139] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Proc. POPL*, pages 232–244. ACM, 2004.

[140] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.

[141] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[142] M. Hocko and T. Kalibera. Reducing performance non-determinism via cache-aware page allocation strategies. In *Proc. ICPE*, pages 223–234. ACM, 2010.

[143] K. Hoder and N. Bjørner. Generalized property directed reachability. In *Proc. SAT*, LNCS 7317, pages 157–171. Springer, 2012.

[144] K. Hoder, L. Kovács, and A. Voronkov. Interpolation and symbol elimination in Vampire. In *Proc. IJCAR*, LNCS 6173, pages 188–195. Springer, 2010.

[145] H. Hojjat, R. Iosif, F. Konecný, V. Kuncak, and P. Rümmer. Accelerating interpolants. In *Proc. ATVA*, LNCS 7561, pages 187–202. Springer, 2012.

[146] A. E. J. Hyvärinen, M. Marescotti, L. Alt, and N. Sharygina. OpenSMT2: An SMT solver for multi-core and cloud computing. In *Proc. SAT*, LNCS 9710, pages 547–553. Springer, 2016.

[147] ISO/IEC JTC1/SC22/WG14. *ISO/IEC 9899:2011: Programming Languages – C*. International Organization for Standardization, 2011.

[148] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based bounded model checking for software verification. *Theor. Comput. Sci.*, 404(3):256–274, 2008.

[149] H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter, and C. Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *Proc. CAV*, LNCS 4144, pages 137–151. Springer, 2006.

[150] JCGM Working Group 2. International vocabulary of metrology – basic and general concepts and associated terms (VIM), 3rd edition. Technical Report JCGM 200:2012, BIPM, 2012.

[151] B. Jeannet, P. Schrammel, and S. Sankaranarayanan. Abstract acceleration of general linear loops. In *Proc. POPL*, pages 529–540. ACM, 2014.

[152] R. Jhala and R. Majumdar. Path slicing. In *Proc. PLDI*, pages 38–47. ACM, 2005.

[153] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4), 2009.

[154] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *Proc. TACAS*, LNCS 3920, pages 459–473. Springer, 2006.

[155] R. Jhala and K. L. McMillan. Array abstractions from proofs. In *Proc. CAV*, LNCS 4590, pages 193–206. Springer, 2007.

[156] D. Jovanovic and B. Dutertre. Property-directed k-induction. In *Proc. FMCAD*, pages 85–92. IEEE, 2016.

[157] N. Juristo and O. S. Gómez. Replication of software engineering experiments. In *Empirical Software Engineering and Verification*, pages 60–88. Springer, 2012.

[158] T. Kahsai, Y. Ge, and C. Tinelli. Instantiation-based invariant discovery. In *Proc. NFM*, LNCS 6617, pages 192–206. Springer, 2011.

[159] T. Kahsai and C. Tinelli. PKind: A parallel k-induction based model checker. In *Proc. Int. Workshop on Parallel and Distributed Methods in Verification*, EPTCS 72, pages 55–62, 2011.

[160] T. Kalibera, L. Bulej, and P. Tuma. Benchmark precision and random initial state. In *Proc. SPECTS*, pages 484–490. SCS, 2005.

[161] D. Kapur, R. Majumdar, and C. G. Zarba. Interpolation for data structures. In *Proc. FSE*, SIGSOFT '06/FSE-14, pages 105–116. ACM, 2006.

[162] E. G. Karpenkov. LPI: Software verification with local policy iteration (competition contribution). In *Proc. TACAS*, LNCS 9636, pages 930–933. Springer, 2016.

[163] E. G. Karpenkov, K. Friedberger, and D. Beyer. JavaSMT: A unified interface for SMT solvers in Java. In *Proc. VSTTE*, LNCS 9971, pages 139–148. Springer, 2016.

[164] E. G. Karpenkov, D. Monniaux, and P. Wendler. Program analysis with local policy iteration. In *Proc. VMCAI*, LNCS 9583, pages 127–146. Springer, 2016.

[165] A. V. Khoroshilov, V. S. Mutilin, E. Novikov, P. Shved, and A. Strakh. Towards an open framework for C verification tools benchmarking. In *Proc. Ershov Memorial Conference 2011*, LNCS 7162, pages 179–192. Springer, 2012.

[166] A. V. Khoroshilov, V. S. Mutilin, A. K. Petrenko, and V. Zakharov. Establishing Linux driver verification process. In *Proc. Ershov Memorial Conference*, LNCS 5947, pages 165–176. Springer, 2009.

[167] G. A. Kildall. A unified approach to global program optimization. In *Proc. POPL*, pages 194–206. ACM, 1973.

[168] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[169] A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke. Automatic abstraction in SMT-based unbounded software model checking. In *Proc. CAV*, LNCS 8044, pages 846–862. Springer, 2013.

[170] F. Kordon and F. Hulin-Hubard. BenchKit, a tool for massive concurrent benchmarking. In *Proc. ACSD*, pages 159–165. IEEE, 2014.

[171] L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In *Proc. CAV*, LNCS 8044, pages 1–35. Springer, 2013.

[172] S. Krishnamurthi and J. Vitek. The real software crisis: Repeatability as a core value. *Commun. ACM*, 58(3):34–36, 2015.

[173] D. Kroening, M. Lewis, and G. Weissenbacher. Under-approximating loops in C programs for fast counterexample detection. *Formal Methods in System Design*, 47(1):75–92, 2015.

[174] D. Kroening and G. Weissenbacher. Verification and falsification of programs with loops using predicate abstraction. *Formal Asp. Comput.*, 22(2):105–128, 2010.

[175] D. Kroening and G. Weissenbacher. Interpolation-based software verification with wolverine. In *Proc. CAV*, LNCS 6806, pages 573–578. Springer, 2011.

[176] O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[177] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT techniques for fast predicate abstraction. In *Proc. CAV*, LNCS 4144, pages 424–437. Springer, 2006.

[178] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Proc. TACAS*, LNCS 2031, pages 98–112. Springer, 2001.

[179] A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *Proc. CAV*, LNCS 7358, pages 427–443. Springer, 2012.

[180] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Syst. Tech. J.*, 38(4):985–999, July 1959.

[181] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proc. LPAR*, LNCS 6355, pages 348–370. Springer, 2010.

[182] J. Leroux, P. Rümmer, and P. Subotic. Guiding craig interpolation with domain-specific abstractions. *Acta Inf.*, 53(4):387–424, 2016.

[183] M. Leucker, G. Markin, and M. R. Neuhäußer. A new refinement strategy for CEGAR-based industrial model checking. In *Proc. HVC*, LNCS 9434, pages 155–170. Springer, 2015.

[184] S. Löwe. CPAchecker with explicit-value analysis based on CEGAR and interpolation (competition contribution). In *Proc. TACAS*, LNCS 7795, pages 610–612. Springer, 2013.

[185] S. Löwe. CPA-RefSel: CPAchecker with refinement selection (competition contribution). In *Proc. TACAS*, LNCS 9636, pages 916–919. Springer, 2016.

[186] S. Löwe, M. U. Mandrykin, and P. Wendler. CPAchecker with sequential combination of explicit-value analyses and predicate analyses (competition contribution). In *Proc. TACAS*, LNCS 8413, pages 392–394. Springer, 2014.

[187] S. Löwe and P. Wendler. CPAchecker with adjustable predicate analysis (competition contribution). In *Proc. TACAS*, LNCS 7214, pages 528–530. Springer, 2012.

[188] S. Lukasczyk. Unbounded heap support for CPAchecker's predicate analysis using SMT arrays. Bachelor's thesis, University of Passau, Software Systems Lab, 2016.

[189] J. McCarthy. Towards a mathematical science of computation. In *Proc. IFIP Congress*, pages 21–28. North-Holland, 1962.

[190] K. L. McMillan. Lazy abstraction with interpolants. In *Proc. CAV*, LNCS 4144, pages 123–136. Springer, 2006.

[191] K. L. McMillan. Interpolants from Z3 proofs. In *Proc. FMCAD*, pages 19–27. FMCAD Inc., 2011.

[192] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *Proc. TACAS*, LNCS 2619, pages 2–17. Springer, 2003.

[193] K. L. McMillan and A. Rybalchenko. Computing relational fixed points using interpolation. Technical Report MSR-TR-2013-6, Microsoft Research, 2013.

[194] C. Meudec. ATGen: Automatic test data generation using constraint logic programming and symbolic execution. *Softw. Test., Verif. Reliab.*, 11(2):81–96, 2001.

[195] J. Morse, L. C. Cordeiro, D. Nicole, and B. Fischer. Handling unbounded loops with Esbmc 1.20 (competition contribution). In *Proc. TACAS*, LNCS 7795, pages 619–622. Springer, 2013.

[196] J. T. Mühlberg and G. Lüttgen. BLASTing Linux code. In *Proc. FMICS/PDMC*, LNCS 4346, pages 211–226. Springer, 2006.

[197] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011.

[198] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proc. ASPLOS*, pages 265–276. ACM, 2009.

[199] T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Unbounded Lazy-CSeq: A lazy sequentialization tool for C programs with unbounded context switches (competition contribution). In *Proc. TACAS*, LNCS 9035, pages 461–463. Springer, 2015.

[200] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[201] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

[202] A. Nutz, D. Dietsch, M. M. Mohamed, and A. Podelski. Ultimate Kojak with memory safety checks (competition contribution). In *Proc. TACAS*, LNCS 9035, pages 458–460. Springer, 2015.

[203] J. Petkovich, A. B. de Oliveira, Y. Zhang, T. Reidemeister, and S. Fischmeister. DataMill: A distributed heterogeneous infrastructure for robust experimentation. *Softw., Pract. Exper.*, 46(10):1411–1440, 2016.

[204] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. Symposium on Programming*, LNCS 137, pages 337–351. Springer, 1982.

[205] Z. Rakamarić and M. Emmi. SMACK: Decoupling source language details from verifier implementations. In *Proc. CAV*, LNCS 8559, pages 106–113. Springer, 2014.

[206] E. F. Rizzi, S. Elbaum, and M. B. Dwyer. On the techniques we create, the tools we build, and their misalignments: A study of KLEE. In *Proc. ICSE*, pages 132–143. ACM, 2016.

[207] H. Rocha, H. I. Ismail, L. C. Cordeiro, and R. S. Barreto. Model checking embedded C software using k-induction and invariants. In *Proc. SBESC*, pages 90–95. IEEE, 2015.

[208] W. Rocha, H. Rocha, H. Ismail, L. C. Cordeiro, and B. Fischer. DepthK: A k-induction verifier based on invariant inference for C programs (competition contribution). In *Proc. TACAS*, LNCS 10206, pages 360–364. Springer, 2017.

[209] O. Roussel. Controlling a solver execution with the RUNSOLVER tool. *JSAT*, 7:139–144, 2011.

[210] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Program. Lang. Syst.*, 27(2):185–235, 2005.

[211] P. Rümmer. *Calculi for Program Incorrectness and Arithmetic*. PhD thesis, University of Gothenburg, 2008.

[212] P. Rümmer and T. Wahl. An SMT-LIB theory of binary floating-point arithmetic. In *Proc. SMT Workshop*, 2010.

[213] J. Sajaniemi. An empirical analysis of roles of variables in novice-level procedural programs. In *Proc. HCC*, pages 37–39. IEEE, 2002.

[214] C. Scholl, F. Pigorsch, S. Disch, and E. Althaus. Simple interpolants for linear arithmetic. In *Proc. DATE*, pages 1–6. IEEE, 2014.

[215] V. Schuppan and A. Biere. Liveness checking as safety checking for infinite state spaces. *Electr. Notes Theor. Comput. Sci.*, 149(1):79–96, 2006.

[216] R. Sebastiani and P. Trentin. OPTIMATHSAT: A tool for optimization modulo theories. In *Proc. CAV*, LNCS 9206, pages 447–454. Springer, 2015.

[217] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proc. FMCAD*, LNCS 1954, pages 127–144. Springer, 2000.

[218] B. Singh and V. Srinivasan. Containers: Challenges with the memory resource controller and its performance. In *Proc. Ottawa Linux Symposium (OLS)*, pages 209–222, 2007.

[219] C. Sinz, S. Falke, and F. Merz. A precise memory model for low-level bounded model checking. In *Proc. Int. Workshop on Systems Software Verification (SSV)*. USENIX Association, 2010.

[220] T. Stieglmaier. Augmenting predicate analysis with auxiliary invariants. Master's thesis, University of Passau, Software Systems Lab, 2016.

[221] A. Stump, C. W. Barrett, D. L. Dill, and J. Levitt. A decision procedure for an extensional theory of arrays. In *Proc. Logic in Computer Science*, pages 29–37. IEEE, 2001.

[222] A. Stump, G. Sutcliffe, and C. Tinelli. StarExec: A cross-community infrastructure for logic solving. In *Proc. IJCAR*, LNCS 8562, pages 367–373. Springer, 2014.

[223] Y.-K. Suh, R. T. Snodgrass, J. D. Kececioglu, P. J. Downey, R. S. Maier, and C. Yi. EMP: Execution time measurement protocol for compute-bound programs. *Software: Practice and Experience*, 47(4):559–597, 2017.

[224] W. F. Tichy. Should computer scientists experiment more? *IEEE Computer*, 31(5):32–40, 1998.

[225] A. van Deursen and L. Moonen. Understanding COBOL systems using inferred types. In *Proc. IWPC*, pages 74–81. IEEE, 1999.

[226] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proc. FSE*, pages 58:1–58:11. ACM, 2012.

[227] J. Vitek and T. Kalibera. Repeatability, reproducibility, and rigor in systems research. In *Proc. EMSOFT*, pages 33–38. ACM, 2011.

[228] A. Volkov and M. U. Mandrykin. Predicate abstractions memory modeling method with separation into disjoint regions. In *Proc. SYRCoSE*, pages 69–73. Institute for System Programming of the Russian Academy of Sciences (ISPRAS), 2017.

[229] T. Wahl. The k-induction principle, 2013. Available at `http://www.ccs.neu.edu/home/wahl/Publications/k-induction.pdf`.

[230] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore. Measuring energy and power with PAPI. In *Proc. ICPPW*, pages 262–268. IEEE Computer Society, 2012.

[231] P. Wendler. CPAchecker with sequential combination of explicit-state analysis and predicate analysis (competition contribution). In *Proc. TACAS*, LNCS 7795, pages 613–615. Springer, 2013.

[232] D. Wonisch. Block abstraction memoization for CPAchecker (competition contribution). In *Proc. TACAS*, LNCS 7214, pages 531–533. Springer, 2012.

[233] D. Wonisch and H. Wehrheim. Predicate analysis with block-abstraction memoization. In *Proc. ICFEM*, LNCS 7635, pages 332–347. Springer, 2012.

[234] I. S. Zakharov, M. U. Mandrykin, V. S. Mutilin, E. Novikov, A. K. Petrenko, and A. V. Khoroshilov. Configurable toolset for static verification of operating systems kernel modules. *Programming and Comp. Softw.*, 41(1):49–64, 2015.

Appendix

A. Machine Architectures with Hyperthreading and Nonuniform Memory Access

As indication how complex and divergent today's machines can be, we show two exemplary machine architectures, which are shown in Figs. A.1 and A.2. In the figures, each CPU is represented by a node labeled with "Socket" and the physical package id, each physical CPU core by a node labeled with "Core" and the core id, and each processing unit by a node labeled with "PU" and the processor id. A processing unit is what a process running under Linux sees as a core. Nodes whose label starts with "L" represent the various caches ("L" for level). Contiguous regions of memory are represented by nodes labeled with "NUMANode" and the node id, and each memory region is grouped together with the CPU cores that it belongs to in a green unlabeled node. The numeric ids in the figures are those that the Linux kernel assigns to the respective unit. The numbering scheme is explained in Sect. 10.4. Such figures can be created with the tool `lstopo` from the Portable Hardware Locality (hwloc) project[1].

Both examples are systems with a NUMA architecture. Figure A.1 shows a system with two AMD Opteron 6380 16-core CPUs with a total of 137 GB of RAM. On this CPU, always two virtual cores together form what AMD calls a "module", a physical core that has separate integer-arithmetic units and level-1 data cache for each virtual core, but shared floating-point units, level-1 instruction cache, and level-2 cache. The cores of each CPU are split into two groups of eight virtual cores. The memory of the system is split into four regions, each of which is coupled with one group of cores. This means that, for example, core 0 can access the memory of NUMANode 0 directly and thus fast, whereas

[1] `https://www.open-mpi.org/projects/hwloc`

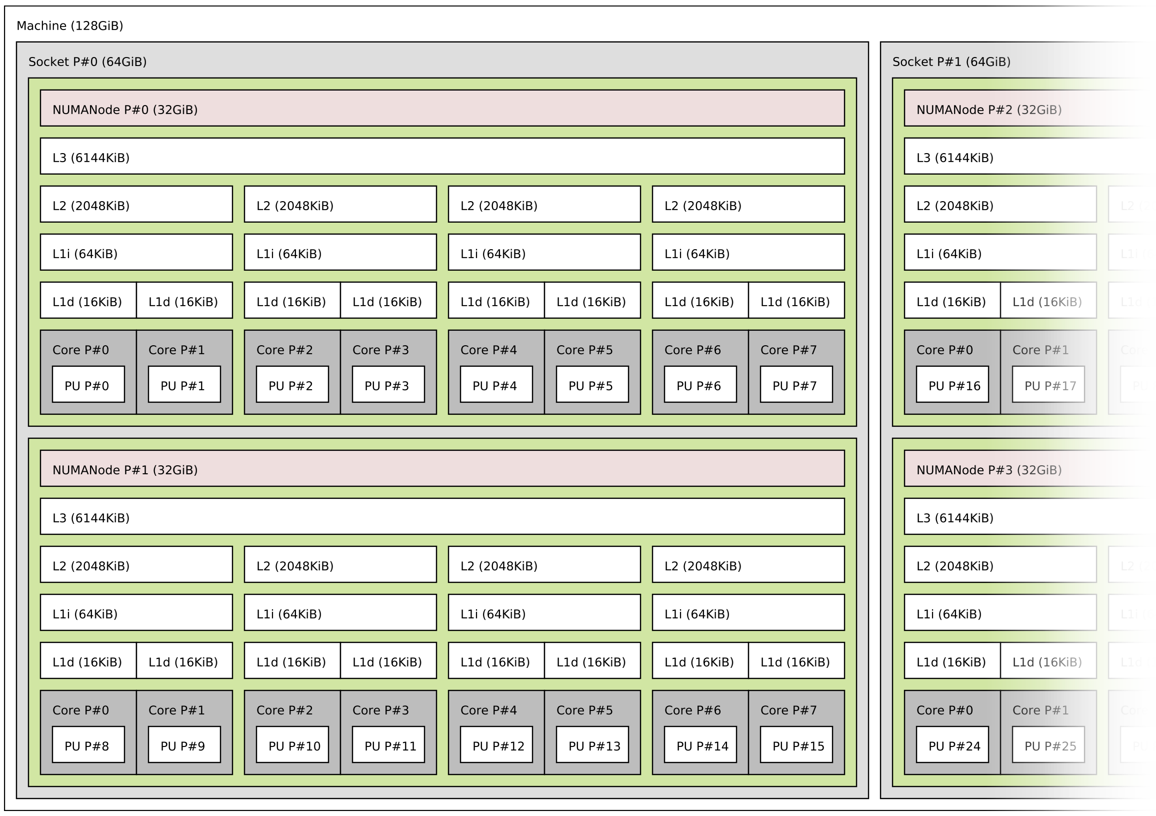


Figure A.1.: Example for a machine with a NUMA architecture:
2 AMD Opteron 6380 CPUs, each with two groups of four modules of two cores and 69 GB (64 GiB) of RAM

accesses to NUMANode 1 would be indirect and thus slower, and accesses to NUMANode 2 and 3 would be even slower as they need to be done via inter-CPU communication channels. Note that on this machine, the two virtual cores of each physical core ("module") got adjacent processor ids and different core ids assigned by the Linux kernel, whereas virtual cores with the same core id on the same CPU actually belong to different physical cores.

Figure A.2 shows a system with two Intel Xeon E5-2650 v2 eight-core CPUs with a total of 135 GB of RAM (caches omitted for space reasons). This CPU model has hyperthreading, and thus there are always two virtual cores that share both integer-arithmetic and floating-point units and the caches of one physical core. The memory in this system is also split into two memory regions, one per CPU. Note that the numbering system differs from the other machine: the virtual cores of one physical core have the same core id, which uniquely identifies a

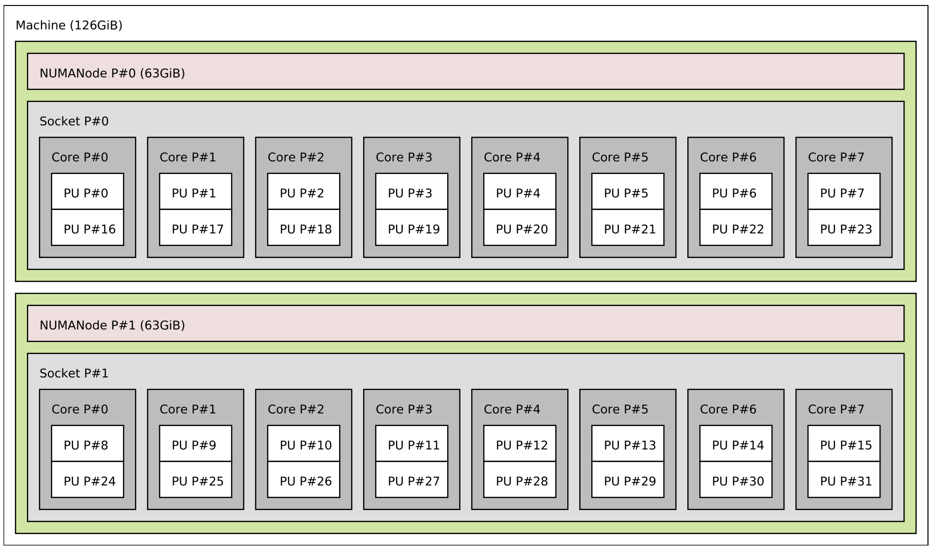


Figure A.2.: Example for a machine with a NUMA architecture:
2 Intel Xeon E5-2650 v2 CPUs, each with eight physical cores with hyperthreading and 68 GB (63 GiB) of RAM

physical core on a CPU here. The processor ids for virtual cores, however, are not consecutive but jump between the CPUs.

Note that both presented systems could appear equal at a cursory glance, because they both have the same number of processing units and approximately the same amount of RAM. However, they differ in their architecture and (depending on the workload) could deliver substantially different performance even if running at the same frequency.

B. Listings

```
from benchexec.runexecutor import RunExecutor
executor = RunExecutor()
result = executor.execute_run(
            args            = [<TOOL_CMD>],
            output_filename = 'output.log',
            hardtimelimit   = 100,
            memlimit        = 16*1000*1000*1000,
            cores           = list(range(0,8)) + list(range(16,24)),
            memory_nodes    = [0])
```

Listing B.1: Example for using module `runexec` from a Python program

```xml
<?xml version="1.0"?>
<!DOCTYPE benchmark PUBLIC
    "+//IDN sosy-lab.org//DTD BenchExec benchmark 1.9//EN"
    "https://www.sosy-lab.org/benchexec/benchmark-1.9.dtd" >

<!-- Example file for benchmark definition, using tool "cpachecker"
     with CPU time limit, memory limit, and 4 CPU cores. -->
<benchmark tool="cpachecker"
           timelimit="900 s" memlimit="8000 MB" cpuCores="4">

  <!-- Define two different configurations to benchmark,
       with different command-line options. -->
  <rundefinition name="predicateAnalysis">
    <option name="-predicateAnalysis"/>
  </rundefinition>

  <rundefinition name="valueAnalysis">
    <option name="-valueAnalysis"/>
  </rundefinition>

  <!-- Global command-line options for all configurations. -->
  <option name="-heap">7000M</option>
  <option name="-noout"/>

  <!-- Define which input files should be used for benchmarking
       (two groups of files declared in separate files). -->
  <tasks name="ControlFlowInteger">
    <includesfile>programs/ControlFlowInteger.set</includesfile>
  </tasks>

  <tasks name="DeviceDrivers64">
    <includesfile>programs/DeviceDrivers64.set</includesfile>
    <!-- These files need a special command-line option: -->
    <option name="-64"/>
  </tasks>

  <!-- Use an SV-COMP property file as specification
    (file ALL.prp in the same directory as each input file). -->
  <propertyfile>${inputfile_path}/ALL.prp</propertyfile>
</benchmark>
```

Listing B.2: Example for an XML file as input for program `benchexec`, defining a benchmark of two configurations of a tool on two sets of files