# Efficient Software Model Checking with Block-Abstraction Memoization

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

eingereicht von
Karlheinz Friedberger
21.05.2021

| | |
|---|---|
| 1. Gutachter: | Prof. Dr. Dirk Beyer |
| 2. Gutachterin: | Prof. Dr. Heike Wehrheim |
| 3. Gutachter: | Prof. Dr. Tomáš Vojnar |
| Tag der Disputation: | 03. 11. 2021 |

# Abstract

The increasing availability of information technology in today's life is a challenge for users as well as for engineers. The execution of a myriad of machine operations per second on each single device as well as world-wide communication between applications in research, industry, and personal domain require reliability and scalability of computer systems, both on hardware and software level. Increasing complexity as well as a big potential for errors make it challenging to develop and maintain such systems. This causes failures in applications, reaching from simple blue screens on personal computers towards accidents in critical infrastructure, such as crashing airplanes and invalid settings for medical equipment, affecting a large number of people.

To address this challenge, it becomes more and more important to use processes to avoid failures in new software and find bugs in already exiting software. Prominent approaches used in quality management of critical systems are formal methods like formal verification and especially model checking, which aims at proving whether a system meets a given specification and fulfills its requirements. Researchers and developers use formal verification for a wide area of applications, including hardware-circuit designs, flight-control systems, and even operating-system drivers that run in everyone's computer. In the last decades, the research community has made several steps towards better usage, adaptation, and understanding of formal techniques. However, due to their high demand of resources in both manpower as well as computational effort, formal methods are still not as used in the standard software development process as plain testing, but almost exclusively in critical applications, where a failure can get life-threatening or disproportionately expensive.

To fulfill the high demand and large-scale availability of quality management, formal verification needs to run automatically and with only small amount of manpower. With our research in this context, we considered the following challenges:

- To enable smaller teams with larger software projects to use formal methods, we need scalable and modular software-verification approaches that not only aim for a lower amount of required resources, but also use the available resources efficiently and allow for caching and reuse of partial analysis results.
- To analyze parallel running software systems with e. g. communicating components, we need a simple formalism and verification technique for parallel systems that works in a domain-independent way, such that several already existing analyses can be applied easily.
- To allow software developers to use model-checking tools regularly, we need to improve the usability and exported data of suchlike, e. g., with using a standardized file format for exchanging analysis results between tools.

- To support the development of new verification approaches and other applications based on formal theory, we need to improve the usability of often-used backend components for solving constraints and analyzing formulas, such as providing simple interfaces and ready-to-use libraries for SMT solvers.

The basis for our research is the open-source verification framework CPAchecker, which provides a convenient context to declare program analyses, supports multiple abstract domains for encoding program behavior, and already contains several approaches and algorithms for software verification.

Our studies on scalable and modular approaches show the potential of block-abstraction memoization, which is a verification technique based on a divide-and-conquer strategy that leverages the internal structure of a program. Using an abstract domain like predicate abstraction, value analysis, or combinations thereof, we evaluate the approach on a large and established benchmark set to provide evidence for its competitive performance against other state-of-the-art approaches and also to show that CPUs with multiple cores can be used to their full potential.

Our research on parallel systems enables interesting insights into multi-threaded verification tasks. We develop a theoretical approach to handle concurrency in a simple, but efficient and also domain-independent way, and provide the corresponding implementation in CPAchecker. Additionally, the established standardized format for the representation of verification witnesses is extended with additional information about the concurrent context and thread scheduling which are valuable artifacts from the verification process.

Orthogonal to the theoretical concepts of software verification, we work on simplifying the use of SMT solvers, which are the basic foundation of several verification approaches. By introducing a common API layer written in Java with bindings towards several distinct SMT solvers, we make it more convenient for the engineer of a verification tool to interact with back-end solvers and even switch between those provide different features and strengths.

# Zusammenfassung

Die zunehmende Verfügbarkeit von Informationstechnologie im Alltag fordert Nutzer und Hersteller gleichermaßen heraus. Für die Ausführung einer Unzahl an Maschinenbefehlen pro Sekunde in jedem einzelnen Gerät und die weltumspannende Kommunikation zwischen Anwendungen in den Bereichen der Forschung, Industrie und Privatleben werden Zuverlässigkeit und Skalierbarkeit von Computersystemen, sowohl auf Hardware- als auch auf Softwareebene, benötigt. Wachsende Komplexität sowie ein großes Fehlerpotential erschweren es, solche Systeme zu entwickeln und dauerhaft zu warten. Dies führt typischerweise zu Fehlern in Anwendungen, von einfachen Bluescreens auf PCs bis hin zu Unfällen in kritischer Infrastruktur, wie z. B. abstürzende Flugzeuge und unpassende Einstellungen an medizinischen Geräten, wovon oftmals eine größere Anzahl an Menschen betroffen ist.

Um dieser Herausforderung zu begegnen, wird es immer wichtiger Prozesse zu verwenden, die Fehler in neuer Software vermeiden und in bereits vorhandener Software finden. Bekannte Ansätze für das Qualitätsmanagement kritischer Systeme sind formale Methoden wie formale Verifikation und insbesondere Model-Checking. Durch Letzteres soll nachgewiesen werden, ob ein System eine gegebene Spezifikation erfüllt und seinen Anforderungen gerecht wird. Forscher und Entwickler setzen formale Verifikation bereits für weite Anwendungsbereiche ein, von hardwarenahen Schaltplänen und Flugsteuerungssystemen bis hin zu Komponenten von Betriebssystemen, die auf unzähligen Computern ausgeführt werden. In den letzten Jahrzehnten hat die Forschungsgemeinschaft schrittweise Anwendung und Bedienung formaler Techniken verbessert und das Verständnis dafür vertieft. Aufgrund ihres hohen Ressourcenbedarfs sowohl an Arbeitskräften als auch an Rechenaufwand werden formale Methoden im Gegensatz zu einfachen Testverfahren immer noch nicht regelmäßig im üblichen Softwareentwicklungsprozess verwendet, sondern fast ausschließlich nur im Rahmen kritischer Anwendungen, in denen Fehler lebensbedrohlich oder unverhältnismäßig teuer sind.

Um die Nachfrage nach Qualitätsmanagement zu erfüllen und die Verfügbarkeit in großem Maßstab zu gewährleisten, muss formale Verifikation weitestgehend automatisch und mit minimaler Nutzerinteraktion durchgeführt werden können. Im Rahmen unserer Forschung haben wir in diesem Zusammenhang folgende Herausforderungen betrachtet:

- Um kleineren Teams die Anwendung formaler Methoden für größere Softwareprojekte zu ermöglichen, brauchen wir skalierbare und modulare Ansätze zur Software-Verifizierung, die nicht nur auf einen geringeren Verbrauch an erforderlichen Ressourcen abzielen, sondern auch die verfügbaren Ressourcen effizient nutzen und das Wiederverwenden von Teilergebnissen der Analyse ermöglichen.

- Für die Analyse von parallel laufenden Softwaresystemen mit z. B. kommunizierenden Komponenten sind ein einfacher Formalismus und eine Verifikationstechnik für parallele Systeme nötig, welche unabhängig von der jeweils eingesetzten Domäne funktionieren, so dass bereits existierende Analysen einfach verwendet werden können.
- Damit Entwickler regelmäßig Model-Checking anwenden können, müssen wir die Benutzerfreundlichkeit der entsprechenden Werkzeuge verbessern sowie beispielsweise Daten für den Austausch von Analyseergebnissen zwischen derartigen Anwendungen in einem standardisierten Dateiformat bereitstellen.
- Um die Entwicklung neuer Verifizierungsansätze und anderer Anwendungen auf Basis von theoretischen Grundlagen zu vereinfachen, sind Benutzerfreundlichkeit und Handhabung häufig verwendeter Backend-Komponenten zu verbessern, die eingesetzt werden, um Gleichungssysteme zu lösen und Formeln zu analysieren, indem wir z. B. einfach zu nutzende Schnittstellen und gebrauchsfertige Bibliotheken für SMT-Solver bereitstellen.

Die Basis für unsere Forschung ist das Open-Source-Projekt CPAchecker, welches ein vorgefertigtes Framework für die Definition von Programmanalysen bietet, mehrere abstrakte Domänen zum Beschreiben von Programmverhalten unterstützt und bereits einige Ansätze und Algorithmen zur Softwareverifikation enthält.

Unsere Studien zu skalierbaren und modularen Ansätzen zeigen das Potenzial der Block-Abstraction Memoization, einer Verifikationstechnik aufbauend auf einer Divide-and-Conquer-Strategie, die die interne Struktur eines Programms nutzt. Auf Basis einer abstrakten Domäne wie Predicate-Abstraction, Value-Analysis oder einer Kombinationen davon bewerten wir diesen Ansatz mit Hilfe einer großen, etablierten Menge an Benchmarks und zeigen seine Wettbewerbsfähigkeit gegenüber anderen Ansätzen auf dem aktuellen Stand der Technik und auch, dass moderne CPUs mit mehreren Kernen voll ausgeschöpft werden können.

Unsere Forschung zu parallelen Systemen ermöglicht interessante Einblicke in die Verifizierung von nebenläufigen Programmen. Wir entwickeln einen theoretischen Ansatz zur Handhabung von Nebenläufigeit auf einfache, aber effiziente und domänenunabhängige Weise und stellen die entsprechende Implementierung in CPAchecker bereit. Zusätzlich wird das etablierte und standardisierte Austauschformat für Fehlerpfade und Beweise bei der Verifizierung mit zusätzlichen nützlichen Informationen aus der Analyse des parallelen Kontexts erweitert.

Unabhängig von den theoretischen Konzepten der Software-Verifikation arbeiten wir daran, die Verwendung von SMT-Solvern zu vereinfachen, welche die zu Grunde liegende Basis mehrerer Verifizierungsansätze sind. Durch die Einführung einer einheitlichen Java-basierten API-Schicht und Anbindungen mehrerer unterschiedlicher SMT-Solver ist die Interaktion mit dem Solver für die Entwickler eines Verifizierungsansatzes bequemer und wir ermöglichen das Wechseln zwischen Solvern für den Zugriff auf deren verschiedene Merkmale und Stärken.

# Acknowledgments

This thesis would not exist without the support of many other people, that was provided during and over that long time. I want to express my deep appreciation to all of them.

First of all, I would like to thank my supervisor Dirk Beyer, who guided me since my early years at university and also throughout this project. He invoked my interest in the theoretical foundations of program analysis and software verification, provided a decent working environment with lots of friendly and helpful colleagues, shared his knowledge, worked on several papers with me, and always had an open ear for discussion.

I also want to thank Heike Wehrheim and Tomáš Vojnar, who have kindly agreed on reviewing this thesis. Additionally, writing papers is not possible without the help of coauthors, which also receive my acknowledgments. I enjoyed working with you, and I have learned a lot from you, although for some of you, it might have been years since our last communication.

I have been working and researching in the middle of helpful colleagues, and I want to express my gratitude to the groups from Passau and from Munich. Both groups provided a decent working environment and it was always a pleasure working with you. Everyone across the office corridor – or lately in the chats, video conferences, and mailing lists – was open for answering questions and fruitful discussions. My special thanks go to Philipp Wendler, who has actually always been available since I started as a student assistant. Along our long common path, I learned a lot of him in the areas of software development, teaching, theory of software verification, executing benchmarks and evaluations, and administrative procedures. Another noteworthy colleague was Matthias Dangl, with whom I shared the office in Passau and Munich and who provided a helping hand and theoretical support whereever required. There are several more colleagues, research and student assistants, older and younger people from university, secretaries, and system administrators, that receive my appreciation.

Additionally, I thank my family, most importantly my parents, for organizational, emotional, and financial support. They are not as deeply familiar with the theoretical concepts of my work – I really tried to explain my topics several times – but provided a convenient and cozy environment for living and working in the Bavarian forest, including space for home office for the last year. Last but not least, my grandmother Anneliese Stöger should be mentioned here, to whom I dedicate this thesis.

# Eidesstattliche Versicherung

Hiermit erkläre ich an Eidesstatt, dass die Dissertation von mir selbstständig, ohne unerlaubte Beihilfe angefertigt ist, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate und gedankliche Übernahmen kenntlich gemacht wurden.

| | |
|---|---|
| **München, 21.05.2021** | **Karlheinz Friedberger** |
| Ort, Datum | Unterschrift |

# Contents


**1. Introduction** **13**

1.1. Motivation . . . 13

1.2. Structure of This Thesis . . . 15

1.3. Software Analysis and Software Model Checking . . . 15

1.4. Frameworks and Tools . . . 16

1.4.1. CPAchecker . . . 16

1.4.2. JavaSMT . . . 17

1.5. Contributions . . . 17

**2. Discussion of Manuscripts** **19**

2.1. Block-Abstraction Memoization . . . 21

2.1.1. Domain-Independent Multi-threaded Software Model Checking . . . 21

2.1.2. In-Place vs. Copy-on-Write CEGAR Refinement for Block Summarization with Caching . . . 22

2.1.3. Domain-Independent Interprocedural Program Analysis using Block-Abstraction Memoization . . . 23

2.2. Multi-Threaded Programs . . . 24

2.2.1. A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAchecker . . . 24

2.2.2. Violation Witnesses and Result Validation for Multi-Threaded Programs 24

2.3. JavaSMT . . . 26

2.3.1. JavaSMT 3: Interacting with SMT Solvers in Java . . . 26

**3. Conclusion and Future Research** **29**

3.1. Summary . . . 29

3.2. Future Work and Prospects in CPAchecker and JavaSMT . . . 30

3.2.1. Block-Abstraction Memoization with More Domains and Further Algorithms 30

3.2.2. Extensions of the Concurrency Analysis . . . 32

3.2.3. JavaSMT . . . 33

3.3. Usage of Verification in Real-World Software Projects . . . 33

**Acronyms** **37**

**Bibliography** **39**

**A. Manuscripts** **49**

Domain-Independent Multi-threaded Software Model Checking . . . . . 51

In-Place vs. Copy-on-Write CEGAR Refinement for Block Summarization with Caching 63

Domain-Independent Interprocedural Program Analysis using Block-Abstraction Memoization . . . . . 83

A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAchecker . 97

Violation Witnesses and Result Validation for Multi-Threaded Programs . . . . . 109

JavaSMT 3: Interacting with SMT Solvers in Java . . . . . 131

# 1. Introduction

## 1.1. Motivation

Communication retrieval and information exchange are an important part of modern society. Within the last decades plenty of computer systems and applications have been developed and are affecting everyone's daily life, scaling from smartphones to computers and data centers and reaching from simple devices to smart homes and industry 4.0. Each typical modern automobile contains about 50 microprocessors, larger vehicles like airplanes or spaceships even multiples of that. All these devices are driven by software and applications that are prone to bugs, which cause, for example, simple bluescreens that can be solved by a reboot or more expensive failures of larger systems like blocked production lines or critical accidents.

Thus, one of the primary goals of developing software is to avoid bugs, i. e., application engineers aim to write non-defective code or to find and then remove bugs from existing software. Testing [97] is the most well-known approach to detect bugs and is applied in private, educational, and industrial software development, but it is not sufficient to detect all problems and to formally prove their absence. Formal methods, such as software verification and model checking, aim to help here by applying manual [69, 78], semiautomated [92, 99], or even fully automated [58, 104] approaches. With larger software systems using millions of lines of code, manual approaches for software verification can become expensive due to high effort and the need of experts for the application domain. Thus, the more efficient way to verify software projects is a fully automated method, which requires less knowledge from the user about formal methods and applies the strengths of available computing resources to solve the task.

The approach of software model checking has successfully been applied in security critical fields, such as the verification of a flood control system [86], automotive control systems [15, 100, 117], avionic software [62], transmission protocols [112], device drivers of operating systems [10, 11, 44, 88], and boot code in computers [60]. Even when compared to testing, formal methods can be competitive in finding bugs [40].

A verification approach models program behavior with an abstract domain that, for example, represents data information, dependencies, or assignments of variables, and reasons about it, possibly by applying a fixed-point algorithm to explore the abstract state space within this abstract domain. The performance of the analysis depends on the used abstract domain and the convergence of the fixed-point algorithm. Popular choices for software-verification approaches are explicit-state model checking [41, 74, 79], SMT-based analyses [26, 39, 50, 94], or approaches based on more specialized data representations like binary decision diagrams (BDDs) [1, 47, 52, 91] or memory graphs [37, 64]. All those different domains have distinct properties and make it challenging to

develop approaches that work in domain-independent level. One successful approach for software model checking is starting with a coarse precision and an abstract view on the program by ignoring certain points of the underlying program, resulting in an imprecise analysis and a lower resource usage. Based on spurious counterexample traces, the precision can then be refined up to a concrete representation of the program [55, 56, 73, 111].

The application of software model checking for developers depends on several factors:

- Performance: How fast are analysis results delivered?
- Validity: Are the reported analysis results correct?
- Usability: How is the analysis result presented towards the user?
- Implementability: How simple is it to write an analysis or extend an existing one?

In an ideal scenario, the performance of an algorithm scales with the available hardware resources, i. e., uses multiple computing units or more memory. For software model checking, there exist *divide-and-conquer* approaches [53, 118] for splitting a larger problem into smaller tasks, solving those separately, and merging the results. This thesis provides insights into the design and implementation of a scalable domain-independent model checking approach based on this idea and compares the analysis different analysis configurations in CPAchecker against each other and with distinct verification tools.

Approaches for model checking for single-threaded programs are well-known and available in several tools. The analysis of programs with multiple threads is more complex, because it requires to take into account a potentially large number of thread interleavings. Verification approaches for those programs exist since several decades [71], however, most applications were limited to specialized domains only, e. g. (symbolic) value analysis [14, 79] or SMT-based analysis [3, 61, 66, 116, 121]. This thesis contains work on a simple domain-independent approach for verifying multi-threaded programs.

Due to coarseness in program modeling, data representation through the used abstract domain, or bugs in the model checking tool, an analysis result can be imprecise and, for example, report a spurious counterexample or an invalid proof. This behavior is unwanted by the user and should be avoided, of course. The user needs to have the possibility to validate all reported data to see the quality of the executed verification approach. This can be shown by validating the certificate (denoted as *witness* [25]), which is automatically generated during the analysis and written in a standardized format. A witness comes in form of either a correctness proof for a valid program that fulfills its specification or a violation witness in case of counterexample traces. It can be imprecise and model not only one concrete, but several program traces at once. A validator takes the witness as input and provides evidence that the analysis result was correct or not. This thesis takes a look at the witness format for exchanging invariants and counterexample traces when applied in the context of multi-threaded programs.

The target user group of software analysis tools consists of developers that intend to write new applications or maintain and extend existing code. One the one hand, developers want to use flexible and efficient tools that do their job without a larger need of configuration. On the

other hand, they want to adapt existing algorithms or invent their own model checking approach to fulfill their requirements. Several aspects of model checking are modeled as constraints or formulas that are given to a constraint-solving library in the backend. To improve the development environment for developing and implementing a new idea, this thesis describes our ready-to-use library for satisfiability modulo theories (SMT) solvers.

## 1.2. Structure of This Thesis

This thesis is a cumulative dissertation and consists of several individual manuscripts. The overall structure is organized as follows:

In this first chapter (Chapter 1), we motivate our topic, provide an introduction to the thesis, define the necessary background, describe the tools, on which our solutions are based on, and present an overview about our contributions.

In the second chapter (Chapter 2), the manuscripts of this cumulative thesis are discussed in detail. We summarize each manuscript, show their connections and relevance in a broader context, and describe the author's contribution for each document.

In the third chapter (Chapter 3), the contributions of this thesis are summarized and future applications and research directions are discussed.

The appendix (Appendix A) provides the published manuscripts in the same form and formatting as they are published.

## 1.3. Software Analysis and Software Model Checking

In the following, a short background of automated software verification is given to provide the necessary foundation for the manuscripts in Appendix A and give insights in the formal definitions of the topic.

The overall topic of this thesis is located in the area of static analysis of software systems, or more concrete, software verification. Static software analysis is the analysis of computer programs without an actual program execution, in contrast to dynamic analysis, which performs the analysis while the program is executing. Static software analysis aims for reasoning over software systems, e. g., for computing additional data like type information or unreachable code, for finding potential bugs, memory leaks, or for detecting other unwanted behavior. It is also applied for proofing the absence of errors by verifying software according to a specification. Typically, the program to be analyzed is available as source code and the analysis can directly access all required information about variables, types, and the control flow.

A subarea of static software analysis is software model checking, which is an approach to check the validity of a program against a specification. The specification describes a liveness or safety property of a system, such as reaching (or not reaching) a certain condition or statement. For theoretical approaches, such as research-related properties, the specification for valid or invalid system behaviour can be given as as a formal description as CTL or LTL [80, 113] or as an automaton, e. g., formalized in the Blast query language [23]. This allows for a flexible

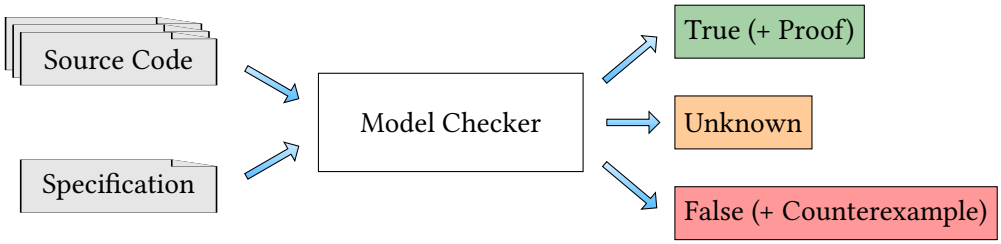



Figure 1.3.1.: Overview of software model checking

and precise modeling of the property. Practical application provides and often uses predefined properties, such as detecting failing assertions in the source code, invalid memory access in low-level programming languages, null-pointer access in object-oriented software, or deadlocks in concurrent software. The behavior of calling external functions and libraries, as well as the system environment in general, can be modeled with a test harness that is part of the specification and which in general depends on the given application and its context. Several predefined properties can be transformed into the corresponding formal description automatically. For more complex cases, such as tracking an execution sequence of function calls, e. g., for locking and unlocking mutexes, some manual effort is required to get a proper formal description.

Software model checking reports either a proof or, in case of a property violation, a counterexample. The analysis of a software system is usually performed by automated tools without manual intervention, due to the potential high effort and computational requirements for larger programs consisting of thousands of lines of code. Figure 1.3.1 gives an overview of the control flow for software model checking. The verification tool (often denoted as *model checker*) takes source code and a specification as input and computes the result, which is either *true* with a proof (if the property is satisfied), *false* with a counterexample (if the specification is violated, e. g., a bug was found), or *unknown* for cases like insufficient resources.

## 1.4. Frameworks and Tools

All manuscripts of this thesis refer to an implementation and an evaluation of the described theoretical concepts. Depending on the manuscript, the corresponding implementation is available in the projects CPAchecker and JavaSMT, for which a short overview is given in the following, including details about their history and development.

### 1.4.1. CPAchecker

CPAchecker [38] is an open-source project for program analysis and software verification.[1] The highly versatile framework is based on the established concept of configurable program analysis [36] and provides a rich set of algorithms and operators for various data representations, e. g., the predicate-based domain [24], explicit values [41], or grounded on specialized data structures like BDD [1, 47, 52, 91] or SMG [64, 98]. The framework does not only support the verification of

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

source code according to a given specification automaton, but also provides an integrated witness validator [25] and several useful features. The verification platform comes with a frontend for C code (which is mainly used), but also provides a frontend for Java and since lately LLVM code. This makes it an ideal playground for the development of new approaches, because many components are provided in a usable manner and can be combined into new analyses quickly and without larger changes in the codebase. CPAchecker is mainly developed by members of Dirk Beyer's group from the Software and Computational Systems Lab [2], including the author of this thesis. The group is currently settled at LMU Munich and previously was located at the University of Passau. The framework is used and extended by researchers and associates from several institutes and universities, including the Institute for System Programming of the Russian Academy of Sciences, the Universities of Paderborn, Darmstadt and Vienna. Variations of CPAchecker regularly participated successfully in the yearly competition on software verification [3] that takes place since 2012 [16, 21] and in the yearly competition on software testing [4] that takes place since 2019 [20, 22].

### 1.4.2. JavaSMT

JavaSMT [9, 85] is an open-source library providing a common Java-based application programming interface (API) for several SMT solvers.[5] The origins of JavaSMT are within CPAchecker, from where the project was split off in 2015 in order to be maintained as independent project. Over the years, not only the list of features and supported backend SMT solvers has grown, but also the group of users and applications based on JavaSMT was expanded. Several developers prefer using this library over the native solver bindings, because of better documentation and easier integration into existing projects.

## 1.5. Contributions

Our contribution consists of several connected topics in the area of software model checking:

This thesis focuses on making existing approaches for software model checking easier to use for developers and scalable for larger systems consisting of multiple components. This is done by modularizing the problem of software model checking into domain-dependent steps and domain-independent operations and algorithms. The domain-dependent steps aim at the encoding and refining of information in a certain theory, e. g., as SMT formulas or as plain mappings of values. The common domain-independent operations and algorithms provide the possibility of handling transfer relation, coverage and termination checks [36]. By separating algorithmic from domain-specific analysis steps, we contribute the possibility of executing independent partial analyses for parts of the program, applying abstraction, caching and reusing intermediate results [118], and parallelization, all together formalized within one verification approach. We contribute a domain-

[2] https://www.sosy-lab.org
[3] https://sv-comp.sosy-lab.org
[4] https://test-comp.sosy-lab.org
[5] https://github.com/sosy-lab/java-smt

independent and interprocedural analysis of programs with (recursive) procedures and different ways for the refinement approach in this context, in order to exploit caching and reusability.

The complexity of software does not only come from its size, but also from the interaction of parallel running applications and their communication. There exist several approaches for the verification of parallel programs, e. g., checking thread- or process-level interleaving with partial order reduction [5, 79], sequentialization [82], or thread abstractions [68, 76]. Our work contributes an efficient and domain-independent approach for the analysis of multi-threaded programs.

A major point for the usability of automated tools is the availability of information about the analysis, such as counterexamples consisting of input arguments and program traces [57], proof witnesses with (loop) invariants [25], or other useful reusable artifacts [35, 42, 49]. The interaction with other tools, e. g., so called *witness validators*, can be done via a standardized witness exchange format based on GraphML, which is an XML-based file format for representing and exchanging graphs.[6] We enhanced this format with the ability to model program traces of concurrent applications, to extend the developer's possibility to access the interleaving of threads that leads to a bug in the program.

For the development of verification approaches and applications based on formal methods, the usage with backend libraries such as SMT solvers is a common solution. To support the developer with a user-friendly interface and to simplify the process of setting up the necessary libraries, we contribute an easy-to-use library JavaSMT for interacting with SMT solvers that includes language bindings for several backend SMT libraries.

Overall, we substantiate the theoretical contributions in the following manuscripts with the corresponding implementations and evaluations as part of the projects CPAchecker and JavaSMT. Our corresponding artifacts are available for further inspection, functional for a replication of the evaluation, and to a great extent reusable for other applications.

[6] https://github.com/sosy-lab/sv-witnesses

# 2. Discussion of Manuscripts

This chapter provides information about the published research manuscripts from Appendix A. Please note that the order of authors follows alphabetic scheme in most cases. We describe the relation between the publications of the thesis author and we give the details on the contributions of the thesis author for each of the manuscripts. The manuscripts are divided into three parts, according to their direction of research.

First, the approach of block-abstraction memoization (BAM) [118] is discussed, which is a divide-and-conquer strategy and aims for improving the performance of a program analysis in a domain-independent way. Therefore, we use the following manuscripts:

- *Domain-Independent Multi-threaded Software Model Checking* [28]
  Authors: Dirk Beyer and Karlheinz Friedberger
  Publication: Proc. ASE 2018
  Discussion: Sect. 2.1.1
  Manuscript: Appendix A, pages 51–61

- *In-Place vs Copy-on-Write CEGAR Refinement for Block Summarization with Caching* [29]
  Authors: Dirk Beyer and Karlheinz Friedberger
  Publication: Proc. ISoLA 2018
  Discussion: Sect. 2.1.2
  Manuscript: Appendix A, pages 63–81

- *Domain-Independent Interprocedural Program Analysis using Block-Abstraction Memoization* [31]
  Authors: Dirk Beyer and Karlheinz Friedberger
  Publication: Proc. ESEC/FSE 2020
  Discussion: Sect. 2.1.3
  Manuscript: Appendix A, pages 83–95

Second, we take a look at our contributions for the verification of multi-threaded programs. Our approach follows a simple pattern and is also independent of the used domain for information representation. Additionally, we extent an established format for violation-witness representation [25] with support for concurrent programs and provide the corresponding validator. The following manuscripts have been published for this part of the thesis:

- *A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAchecker* [27]
  Authors: Dirk Beyer and Karlheinz Friedberger
  Publication: Proc. MEMICS 2016
  Discussion: Sect. 2.2.1
  Manuscript: Appendix A, pages 97–107

- *Violation Witnesses and Result Validation for Multi-Threaded Programs* [34]
  Authors: Dirk Beyer and Karlheinz Friedberger
  Publication: Proc. ISoLA 2020
  Discussion: Sect. 2.2.2
  Manuscript: Appendix A, pages 109–130

Third, we do not only work on applying software model checking to programs and developing approaches to do that efficiently, but also take a look at a lower level, namely the interaction between a model checking framework and the underlying SMT solver. We developed and maintain an API to interact with several different SMT solvers in the backend, based on a binary interface, i. e., including direct operations for memory manipulation and formula transformation. The description is available in the following manuscript:

- *JavaSMT 3: Interacting with SMT Solvers in Java* [9]
  Authors: Daniel Baier, Dirk Beyer, and Karlheinz Friedberger
  Publication: Proc. CAV 2021
  Discussion: Sect. 2.3.1
  Manuscript: Appendix A, pages 131–144

## 2.1. Block-Abstraction Memoization

### 2.1.1. Domain-Independent Multi-threaded Software Model Checking

The article *Domain-Independent Multi-threaded Software Model Checking*, which is reprinted in Appendix A, pages 51–61 of this dissertation, was authored by Dirk Beyer and Karlheinz Friedberger, and published by ACM in the Proceedings of ASE '18, pages 634–644 [28] with an additional artifact [30] containing the corresponding software, benchmarks, and results.

The article describes an approach for program analysis that applies BAM [118] in a parallel manner to leverage multi-core shared-memory machines.

BAM applies a *divide-and-conquer* strategy and splits a program into smaller components, such as loop or function blocks. Each program block is then analyzed by a separate sub-analysis and its resulting summary, denoted as *block abstraction*, is cached for later usage, such as embedding it into the surrounding context. The framework CPAchecker allows to exchange the algorithm for the sub-analysis and, for example, apply a plain reachability analysis or counterexample-guided abstraction refinement (CEGAR). The idea of the published article is an extension of this approach with the intention to analyze those blocks in parallel and apply independent sub-analyses asynchronously, in order to use all available resources on a multi-core machine. For this approach, dependencies between program blocks need to be considered, such as connections based on plain data flow or function calls. The overall approach is independent of the underlying domain and works for, e. g., value, interval, or predicate domain and even combinations thereof. The experimental evaluation in the manuscript uses value analysis [41], shows no noticeable overhead in runtime for verification tasks that do not benefit from a parallel analysis, and provides evidence for performance improvement for several tasks where independent branches are explored during the analysis.

The contribution of this article consists of two parts: First, the theoretical contribution is a formal definition of the extended definition of BAM, i. e., with the ability to distribute its workload across several threads. Second, the implementation into the framework CPAchecker allows to use and evaluate the approach for a large set of benchmark tasks. The emphasis is on providing a solution that follows the principle of separation of concerns: making a program analysis benefit from multiple processing units is orthogonal from the problem of designing and implementing the abstract domain and components for the underlying program analysis.

Karlheinz Friedberger is the main author of the article and responsible for more than 70 % of the article's contents. His contribution includes

1. the conception and description of the presented approach with a focus on reusing components like the configurable program analysis (CPA) concept and existing algorithms,
2. the development of the presented implementation in the framework CPAchecker based on the existing implementation of BAM and the CPA concept, and
3. the experiment execution and discussion of the evaluation.

### 2.1.2. In-Place vs. Copy-on-Write CEGAR Refinement for Block Summarization with Caching

The article *In-Place vs. Copy-on-Write CEGAR Refinement for Block Summarization with Caching*, which is reprinted in Appendix A, pages 63–81 of this dissertation, was authored by Dirk Beyer and Karlheinz Friedberger, and published by Springer in the Proceedings of ISoLA '18, pages 197–215 [29].

The article compares two different refinement strategies that can be applied in the context of BAM [118]. Whenever CEGAR instructs the program analysis to refine the abstraction along a program path, several block summaries are affected and need to be updated with the refinement information. The analysis can choose between either a destructive *in-place* strategy that modifies existing block abstractions or a constructive *copy-on-write* approach where existing data remains unchanged.

BAM applies a *divide-and-conquer* strategy and splits a program into smaller components, such as loop or function blocks, in order to compute summaries, denoted as block abstractions. For performance, BAM aims to apply the same block abstraction at multiple program locations, e. g., if the same function call appears several times in the program, its summary can be used for each of the call locations. A program path typically traverses several blocks. When applying CEGAR in the analysis, the refinement step needs to remove those parts of the abstract reachability graph (ARG) that contain imprecise information and updates the corresponding abstract states accordingly. Deleting an existing block abstraction removes the corresponding abstract states at all program locations, where this block abstraction was applied. This is denoted as destructive *in-place* refinement and it might also affect unrelated parts of the state space, which need to be re-computed in the later analysis. The alternative approach uses a constructive *copy-on-write* refinement, where only the (minimal) local part of the ARG is modified and existing unrelated parts remain untouched. The evaluation in the manuscript compares both strategies using predicate analysis [39] and value analysis [41] as underlying domains. For a large set of benchmark tasks, the results show no noticeable difference. However, in case of the new *copy-on-write* refinement, the exported data for the user, such as proof information and violation witness, is always available, whereas this kind of information can be incomplete when using the *in-place* strategy.

The contribution of this article consists of three parts: First, the theoretical contribution is a formal definition of the *copy-on-write* refinement step in BAM. Second, the implementation into the framework CPAchecker allows to use and evaluate the approach for a large set of benchmark tasks. Third, the evaluation compares the existing *in-place* and the new *copy-on-write* approach on a large set of benchmarks.

Karlheinz Friedberger is the main author of the article and responsible for more than 70 % of the article's contents. His contributions were

1. the definition and description of the presented approach
2. the implementation of the presented refinement in the framework CPAchecker based on the existing implementation of BAM, CEGAR and the CPA concept, and
3. the comparison of the refinement approaches and discussion of the experimental evaluation.

### 2.1.3. Domain-Independent Interprocedural Program Analysis using Block-Abstraction Memoization

The article *Domain-Independent Interprocedural Program Analysis using Block-Abstraction Memoization*, which is reprinted in Appendix A, pages 83–95 of this dissertation, was authored by Dirk Beyer and Karlheinz Friedberger, and published by ACM in the Proceedings of ESEC/FSE '20, pages 50–62 [31] with an additional artifact [33] containing the corresponding software, benchmarks, and results.

The article describes an interprocedural program analysis based on BAM [118], which applies procedure summarization to analyze programs with procedure calls. The focus is the analysis of recursive programs, which can not be analyzed by conventional interprocedural analyses and require special attention due to the potentially endless unrolling of recursive procedures and potential naming collision when encoding program variables.

The interprocedural analysis splits a program into procedure blocks and computes procedure summaries for them, which aligns with the definition of function blocks and block abstractions in BAM. Every procedure block is analyzed independently and the corresponding abstraction is then applied in the surrounding context. For recursive programs, a fixed-point algorithm terminates the recursion if every procedure is sufficiently unrolled. The article gives detailed insights for this approach, including a definition of the necessary changes towards BAM itself, such as the fixed-point algorithm to terminate recursion in a sound manner and modifications to operators to avoid colliding names in the encoding of program variables from different scopes. The algorithm is independent of the underlying domain and works for, e. g., value, interval, or predicate domain and their combinations. The manuscript contains an experimental evaluation using several domains and shows that the analysis of recursive tasks is performing well when compared to other participants of SV-COMP '20 [19].

The contribution of this article consists of two parts: First, the theoretical contribution contains a formal definition of the extended definition of BAM, including the fixed-point algorithm and operators to work with recursive procedure calls. Second, the implementation into the framework CPAchecker allows to apply the approach for a large set of to real-world benchmark tasks. The approach follows the principle of separation of concerns: it makes interprocedural program analysis independent of the underlying domain by defining common algorithms and operators.

Karlheinz Friedberger is the main author of the article and responsible for more than 70 % of the article's contents. His contributions are

1. the conception and description of the presented approach with a focus on reusing components like the CPA concept and existing algorithms,
2. the development of the presented implementation in the framework CPAchecker based on the existing implementation of BAM, and
3. the execution of experiments and the discussion of the evaluation.

## 2.2. Multi-Threaded Programs

### 2.2.1. A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAchecker

The article *A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAchecker*, which is reprinted in Appendix A, pages 97–107 of this dissertation, was authored by Dirk Beyer and Karlheinz Friedberger, and published by EPTCS in the Proceedings of MEMICS '16, pages 61–71 [27].

The article describes a simple and efficient approach for verifying multi-threaded programs that is implemented in the framework CPAchecker and is purely based on the CPA concept.

This article extends the reachability algorithm of CPAchecker with a new and simple CPA for tracking multiple program locations per abstract state based on multiple threads in a program task. The program analysis is orthogonal to the existing data-flow analysis and works on a domain-independent level. Thus, it can be applied with, e. g., a BDD-based, interval, value, or predicate analysis, or combinations thereof. The approach features several possibilities for optimization and was intended as a playground for developers and researchers. The article provides an evaluation for different domains and also other comparable tools, such as Cbmc and Vvt.

The contribution of this article consists of the following parts: First, the theoretical contribution is a formal definition of a program analysis for multi-threaded programs within the CPA context, i. e., a description of the required operators and components of the framework that can be combined with this approach. Second, the implementation into CPAchecker allows to easily use, extend, and evaluate the approach, and it provides a stable foundation for research.

A follow-up research project was the development of a validator for violation witnesses coming from multi-threaded programs, as discussed in Sect. 2.2.2 and described in the corresponding manuscript in Appendix A. The absence of eager optimization in our approach allows the validation of violation witnesses from other tools like Cbmc, Divine, Esbmc, or Lazy-CSeq without interfering or conflicting with their own optimization strategies.

Karlheinz Friedberger is the main author of the article and responsible for more than 70 % of the article's contents. He contributed

1. the conception and description of the presented approach with a focus on reusing components like the CPA concept, search strategies, and existing algorithms,
2. the implementation in the framework CPAchecker using existing components like the CPA concept or the algorithm for state-space exploration, and
3. the experimental evaluation of the approach using different abstract domains and against other tools.

### 2.2.2. Violation Witnesses and Result Validation for Multi-Threaded Programs

The article *Violation Witnesses and Result Validation for Multi-Threaded Programs*, which is reprinted in Appendix A, pages 109–130 of this dissertation, was authored by Dirk Beyer and Karl-

heinz Friedberger, and published by Springer in the Proceedings of ISoLA '20, pages 449–470 [34] with an additional artifact [32] containing the corresponding software, benchmarks, and results.

Validating results from a previous verification run is an important topic in the research community in order to gain trust in verification results and concretize reported counterexample information. The article describes an extension of the existing GraphML-based format for verification witnesses [25] to include the validation of error traces in multi-threaded programs. The reference implementation is available in the framework CPAchecker.

The existing GraphML-based format was invented to exchange proof and violation information [17, 35] such as invariants, state space guards, or variable assignments between verification tools. The extension was implemented and applied in CPAchecker for SV-COMP '17 [17], where developers of several other tools agreed on the usefulness [1] and adopted their tools accordingly. The formal definition was published in our manuscript [34]. The article defines an extension for the format by introducing information about the interleaving of multiple threads of a program, which is essential for validating error traces from violation witnesses with decent performance, e. g., instead of requiring the exploration of a potentially large state space of the program. We identified the following information as key points and defined them for the standard: the creation point of a new thread and the currently active thread for each statement along a program trace. The reference implementation of the witness validator was evaluated on the violation witnesses from the SV-COMP '20 [2].

The contribution of this article consists of the following parts: First, the GraphML-based format for witnesses is extended by the required thread-related information. Second, the reference implementation into CPAchecker gives the possibility to evaluate the approach. Third, the experimental evaluation showed that several verification tools already provide the necessary information in their witnesses and that the performance of the validation of violation witnesses is improved by the available thread information.

Karlheinz Friedberger is the main author of the article and responsible for more than 70 % of the article's contents. He contributed

1. the overall conception and description of the presented approach with a focus on keeping changes to the GraphML-based format minimal,
2. the implementation in the framework CPAchecker that is not using any eager optimization like partial order reduction and allows to validate multiple violation witnesses from other tools, and
3. the experimental evaluation of the approach using different abstract domains and against other tools.

The initial design decisions came from Matthias Dangl, who gave not only hints on the required information, but also helped with developing and debugging the initial validation executions.

[1] https://github.com/sosy-lab/sv-witnesses
[2] https://sv-comp.sosy-lab.org/2020

## 2.3. JavaSMT

### 2.3.1. JavaSMT 3: Interacting with SMT Solvers in Java

The article *JavaSMT 3: Interacting with SMT Solvers in Java*, which is reprinted in Appendix A, pages 131–144 of this dissertation, was authored by Daniel Baier, Dirk Beyer, and Karlheinz Friedberger, and published by Springer in the Proceedings of CAV '21, pages 195–208 [9]. The additional artifact [8] contains the corresponding software, benchmarks, and results.

Theorem provers and SMT solvers are more and more integrated in applications like software verification. The article is an updated version of a previous article [85] from 2016 and is about a common Java API for several SMT solvers, which acts as an intermediate layer in the software architecture between the user application and the backend SMT solver. It aims to avoid the lock-in on a specific SMT solver. The library includes the Java native interface (JNI) bindings for several solvers written in a low-level language like C. The corresponding project JavaSMT [3] is based on code extracted from CPAchecker and has been actively developed and extended over the last years.

The interaction with an SMT solver can be done in two ways: either via the String-based SMTLIB standard [12] or via a binary API using native solver bindings. From runtime and usability perspective, the interaction with backend libraries via a binary API is often preferred by developers. Additionally, it provides type-safe usage (in Java this is checked at compile time) and proper error handling, e. g., via exceptions. If an application uses the API of a specific SMT solver directly, there is a lock-in effect and switching to a different SMT solver in the future can be difficult and error-prone. Switching the SMT solver can be required if an SMT solver does not (or no longer) contain a required feature or SMT theory, or if another SMT solver performs better in runtime or memory consumption. JavaSMT provides not only common methods to create, but also to manipulate formulas for several SMT solvers. This includes traversing formulas efficiently using the visitor design pattern.

Several SMT solvers already serve bindings for some programming languages, like the language they are implemented in. Most SMT solvers are written in C/C++, so interaction in this low-level language is often the simplest way. The support for higher-level languages is sparse. A common language binding for several SMT solvers is Python, as it directly allows the access to C code and avoids automated memory management operations like asynchronous garbage collection. Bindings for Java are available for some SMT solvers, such as Z3 and MathSAT5, but missing, unsupported, or unmaintained for others, such as CVC4 and Boolector. There exist similar APIs like PySMT [70], Smt-Switch [93], metaSMT [106], ScalaSMT [54], and Scala SMT-LIB [51]. They are based on different programming languages and offer a different set of features.

A former article about JavaSMT was published in 2016 [85] and describes an older state of this project with the initial design and features. For 2021, this old article contains already outdated information, and thus an updated article was published, which is reprinted in Appendix A, pages 131–144. The new article contains the SMT theories and features for each SMT solver integrated in JavaSMT. The development over several years also brought changes, for exam-

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

ple, JavaSMT got support for more SMT solvers, such that now Boolector [4], CVC4 [5], and Yices [6] can directly be used, too.

The contribution of this article consists of two parts: First, the architecture and implementation of JavaSMT is presented. The basic structure of the application is divided into formula managers for building formulas in various theories and prover environments for querying the SMT solver, e. g., for satisfiability checks, models, interpolants, and unsat cores. Second, an experimental evaluation provides data about different runtime behavior of three verification algorithms (bounded model checking (BMC), *k*-Induction, and predicate abstraction) implemented in CPAchecker when using distinct SMT solvers via JavaSMT. Overall, there is a noticeable difference between several SMT solvers in performance, i. e., as a consequence of maturity and optimization of the solvers, but also depending on the SMT query based on the program under analysis.

Karlheinz Friedberger is a co-author of the article and responsible for about 50 % of the article's contents. He contributed

1. the implementation of the solver bindings towards Princess, SMTInterpol, and Z3,
2. large parts of the integration of JavaSMT into CPAchecker, and
3. the documentation of solvers, their supported theories, and their features.

Daniel Baier contributed to the codebase of JavaSMT, e. g., the integration of Boolector [7]. He is responsible for about 30 % of the article's contents, including the data and design of tables and figures in the manuscript. The initial design decisions for JavaSMT came from Philipp Wendler, who guided the initial development, provided support for the depending components in the framework CPAchecker and the infrastructure for releasing JavaSMT as a library. The further development of the project was organized by Karlheinz Friedberger, who initiated the integration of more SMT solvers [7], as well as support for a broader range of build frameworks and operating systems.

[4] https://boolector.github.io
[5] https://cvc4.github.io
[6] https://yices.csl.sri.com
[7] The SMT solvers Boolector and Yices 2 were integrated into JavaSMT as part of bachelor theses at the LMU Munich [7, 101]

# 3. Conclusion and Future Research

In the following, we summarize the conclusions that can be deduced from the manuscripts, where we have investigated several approaches for making software model checking more modular and scalable for larger systems or easier to use and extend for developers.

## 3.1. Summary

Overall, the conclusions is split into three topics, in the same manner as previous parts of this thesis.

First, we have taken a deeper look on BAM, which provides a domain-independent approach for software analysis based on a divide-and-conquer strategy and caching. Our research shows several points for improvements and beneficial changes. We extend the modular approach with a fully interprocedural analysis and demonstrate a benefit when verifying recursive procedures. With the default refinement strategy suffering from several smaller issues (such as not being able to report complete proofs for several program tasks), we present an alternative refinement strategy of block abstractions based on a copy-on-write strategy that comes with no additional performance costs. Applying parallel computation effort (such as multiple CPU cores) gives performance improvements for a wide range of benchmark tasks, while this does not add an overhead for tasks that are not affected by this extension. Overall, our research shows that BAM can improve the scalability of software analysis and is a good foundation for further extensions.

Second, our contribution contains the analysis and verification of multi-threaded programs in CPAchecker and the GraphML-based exchange format for violation witnesses for multi-threaded programs. Our lightweight approach can be combined with several domain-specific analyses and is applied in the context of program verification and witness validation. From a user's point of view, the specification and availability of violation witnesses for multi-threaded programs is already a benefit, but the support for their validation is (up to now) unique for CPAchecker. The only available witness validator for multi-threaded programs is CPAchecker (or an application that is based on it [45]) and it was regularly applied in SV-COMP [1] since 2018 [18, 19, 21]. The given manuscripts show the competitive performance of our verification approach against other model checking tools, such as Cbmc and Vvt.

Third, the design and the development of our Java-based API for SMT solvers results in the project JavaSMT, which is the basis of all SMT-based analyses in CPAchecker. In contrast to a plain SMT-LIB-based interaction with the SMT solver, it provides a type-safe and user-friendly way to create and analyze queries for several included backend solvers. This not only shows a deep

[1] https://sv-comp.sosy-lab.org

understanding of the application domain, but has also attracted several developers that prefer to use our library interface over the SMT-LIB-based solver interaction. Currently, there exist several applications and research publications based on our library [81, 84, 110].

## 3.2. Future Work and Prospects in CPAchecker and JavaSMT

Our contribution consists of several steps towards the improvement of software verification in terms of scalability, modularity, and user-friendliness. However, a broad application and acceptance of formal methods in industry still needs some more effort, because the required domain-specific knowledge for users is still high, and simpler approaches for quality assurance like testing can be applied much faster. Depending on our projects, as well as independently of them, we need to look for possibilities to make the objective of software verification approachable to a broader audience. As the team around CPAchecker and JavaSMT is growing, more opportunities evolve, new ideas appear, and the group of developers and users is expanding. In the following, we will discuss several topics that appear to be most important for further research related to the area of this thesis.

### 3.2.1. Block-Abstraction Memoization with More Domains and Further Algorithms

The provided manuscripts define the operators of BAM for several abstract domains (such as predicate, interval, and explicit value domain) and algorithms (mainly the application of the CPA algorithm within a CEGAR loop). This definition is of course not conclusive and can be extended.

#### 3.2.1.1. Domains

There are more abstract domains, such as symbolic memory graph (SMG) [64] or octagon [96], that can benefit from memoization and a reduction operation for abstract states, to make their application scalable for larger programs. The reduction and expansion technique for SMG can be similar to the approach used for the explicit value domain, i. e., by removing assignments (and heap-related data) from program scopes that are currently hidden by a more local scope. The octagon domain tracks relations between variables, such that any reduction approach needs to take dependencies between variables into account.

Additionally, instead of using a homogeneous domain for all blocks of a program, combining different domains in different block abstractions might bring new insights about a program. The possibility to choose the right domain per block might lead to another direction of scalability, such that, e. g., smaller blocks are analyzed with a more concrete domain, loop blocks are summarized with a domain specialized for loops, and larger blocks are considered by a domain that allows to abstract from a concrete and expensive state space representation.

Translating information between different abstract domains is still an unsolved difficulty, because of a potential information loss and incompatibility of data representations. To name an example, the translation from explicit value domain into predicate domain is straight-forward and

can be done by encoding all assignments of variables to their value as equalities in a conjunctive formula. The way back, i. e., from an arbitrary formula to a simple mapping of variables and their values, causes a loss of information in general. The reason for this asymmetry lies in the expressiveness of the domains, where the predicate domain can apply all Boolean operations and explicit values just use equalities.

#### 3.2.1.2. Algorithms

The concept of BAM is domain-independent in general. We will now take a look on existing domain-independent approaches that it can be combined with, such that the complete analysis might benefit.

The concepts of CEGAR and CPA are available, and (seen from the view of the analysis) CEGAR always wraps the complete analysis. Whenever a refinement is required, the nested analysis, e. g., an instance of the CPA algorithm, needs to interrupt, such that a counterexample trace from the program root towards the recently found program location of property violation can be refined. Aiming at a strategy for even lazier refinement, more locality and an improved modularity of the involved components, a deeper integration of CEGAR into this approach could be proposed, e. g., such that refinements are already applied within the local block instead of global level. This would align with the idea of refinement selection [43], but considers only the suffix of a possible counterexample trace to be involved in the refinement. Within a framework like CPAchecker, even a combination of distinct algorithms like *k*-Induction and predicate analysis in block-wise manner could be possible.

The idea of having multiple threads computing parts of the same program analysis [118, 120] can be extended to distribute the workload onto separate processes, maybe even on distinct hosts [79] or in a cloud-based environment. The biggest complexity lies in the fact that there is no shared memory in such architectures and that the analysis needs to perform communication between separate instances. During the development of the manuscript Appendix A, pages 51–61, there was already a prototype [105] for such a system, based on the Akka framework for workload distribution [2], which did not yet meet the expectations of a fast and scalable verification technique, due to the expensive usage of network communication. However, perhaps this idea can be targeted again in the future.

There are several SMT-based algorithms available [26], and for some of them, an interaction with BAM might be possible. The tool Whale [2] extended Impact [48, 95] into an intra-procedural analysis. Additionally, HiFrog [4] uses SMT-based procedure summaries and interpolation. The integration of such an approach into BAM could show the differences to plain block abstraction, and a further comparison with predicate abstraction [39] would be possible.

A modular software-analysis approach requires several axes of scalability for exploring the abstract state space:

- *horizontal scaling*
  divides the frontier of the exploration algorithm onto several computation units.

[2] https://akka.io

- *vertical scaling*
  allows to analyze intermediate parts of the state space without the current context.
- *domain-specific scaling*
  aims for abstraction and can reduce the number of explored states in general.

Domain-independent approaches target for horizontal and vertical scalability by dividing the analysis into separate parts. This aligns with the design and implementation of BAM, where the exploration of program blocks and computing block abstractions is however bounded due to dependencies between program blocks. We provide evidence for horizontal scaling with BAM in manuscript Appendix A, pages 51–61, where a multi-threaded approach is described that computes independent block abstractions separately. With the work on manuscript Appendix A, pages 83–95 we provide the first step for analyzing blocks without their context, which aims towards vertical scaling. The approach still suffers from some remaining dependencies between blocks, when it comes to data-flow analysis. We presented an improved interpolation strategy using tree interpolants [75] that allows to independently analyze procedures in a modular way. Similar strategies could be developed for other domains, but some data-flow dependencies will still remain. Domain-specific scaling is defined by the usage of the domain. In the context of CEGAR, the precision enables good control over the granularity of the analysis. In general, SMT-based approaches tend to have a more compact representation of the state space than, e. g., an explicit-value domain. The performance of solving SMT queries competes with the memory usage for a larger state-space representation. CPAchecker supports the combination of abstract domains to benefit from their strengths [115]. Further research in that direction might choose the precision and abstract domains depending on the program block to be analyzed.

### 3.2.2. Extensions of the Concurrency Analysis

Verifying concurrent programs has been a topic of research since many years, e. g., with the development of partial order reduction [5, 67], decomposition techniques [6, 65] and sequentialization strategies [66, 90, 103]. The approach presented in Appendix A, pages 97–107 is lightweight and does not apply a fully fledged partial order reduction technique, but only a minimal strategy to gain acceptable performance without losing precision. The extension with some form of partial order reduction technique [71], such as computing either stubborn sets, ample sets, or persistent sets, is an obvious first choice for further development for the verification approach. The witness validation of property violations in multi-threaded programs needs to stay as general as possible, in order to accept witnesses from different tools. Here an eager optimization like partial order reduction or limiting the potential state space through an extension of the analysis itself is riskful, because it is not obvious which strategy was applied for producing a given witness by another model checker. A potential performance overhead or – worst case – an unsound analysis that delivers a wrong answer must be avoided.

An open research topic are correctness witnesses for multi-threaded programs. The main issues thereby are a potentially unbounded number of threads and the specification of invariants for cross-thread data dependencies. The first issue comes from the bounded nature of the witness

automaton, that can model loops, but not a potentially infinite number of concurrent contexts. The second issue is related to the encoding of program variables and might be solvable with an additional flag per variable that denotes the corresponding thread context. Additionally, the representation of invariants and assumptions as expression in the language C is not suited, e. g., for specifying quantified constraints. This expressiveness is available in other verification frameworks, such as Frama-C [13].

Our verification approach is not limited to analyze the reachability of function calls or error labels in the code, so more advanced safety properties can be specified, for example, to verify the reachability of deadlocks and race conditions. The analysis of liveness properties with our lightweight approach for state-space exploration also remains an open topic.

The manuscript Appendix A, pages 97–107 describes an approach that is independent of the underlying domain and applies a BDD-based domain and explicit values in the evaluation. Over the years, CPAchecker was enriched with support for SMT-based analysis of multi-threaded programs [89], such that BMC as well as predicate analysis can be used for the verification of concurrent programs. Prospective work might include further domains, such as SMG-based analysis, for analyzing memory-related properties, including features like weak and strong memory modeling, different machine architectures, or explicit communication between threads in a channel-like manner.

### 3.2.3. JavaSMT

Since the first public release of JavaSMT in 2015, the project was extended with more SMT solvers, theories, and support for more architectures and build systems. In the future, new features could be requested, like the update for an existing SMT solver, and the support of a new one or a currently unsupported theory required by a user. As this project serves as a library that performs its work in the background of other applications, such as CPAchecker, we do not expect larger steps in its development, but aim to a stable and well-maintained API that remains usable and stable over several years of development and stays up-to-date with further research in the area of SMT solvers.

## 3.3. Usage of Verification in Real-World Software Projects

The previously given description of prospects is mainly about existing projects and their perspective. This section considers a broader audience, gives an outlook on the difficulties for software verification applied to real-world projects, and possible steps (also as a result of this thesis) towards a solution, such that software verification becomes a standard process in the development of larger computer and communication systems.

Up to now, mostly specialized (research) teams in certain branches of industry apply software verification, in cases where applications are critical regarding security, reliability, and trustworthiness. Aside of specialized projects for avionic or automotive software [15, 62, 100, 112, 117], further prominent examples for such undertakings are the Linux Driver Verification (LDV)

project [3] [44, 88] (where CPAchecker is applied successfully to verify Linux kernel drivers), the Static Driver Verifier (SDV) project [4] [11] (analyzing kernel-mode drivers in Microsoft Windows), or projects for secure communication software [5] (focused on the HTTPS ecosystem and TLS protocol) and verified boot code software [60] (CBMC applied for software in AWS data centers). Overall, companies applying formal methods are mostly targeting large and existing code bases and a way to minimize interruptions for the developer [102].

Additionally, verification tools tend to report a quantity of false alarms, when applied for large existing code bases that have not yet been formally analyzed before. Ideas like incremental verification [53, 77, 107, 109] help with detecting errors in code that was modified recently, while not reporting potential bugs for unchanged existing code. With applying an incremental approach, the developer does not need to take a look at bug reports for untouched parts of the program under analysis, and thus, is not further interrupted in his workflow. The basic idea behind incremental verification is a modular analysis, e. g., such that information like constraints or partial analysis results can be cached, stored in a database, and reused in the further verification process [35, 49, 83, 114, 119]. An example for such an analysis is the described approach of BAM, where abstractions of program blocks (like loops or functions [31]) are reused whenever a program block depending on some program context needs to be analyzed multiple times. This allows for a parallel application of the analysis [28] and brings additional performance. Overall, scalability and a quick response time for the developer are a main goal of our work.

One of the largest inhibition thresholds of applying software verification in industry and research comes from the high initial costs and domain-specific knowledge for getting started, the required manpower and computational resources for successfully analyzing larger software projects, and possibly the rapid changes in software, both in the applications to be verified and in the verification engines themselves. With the research on witnesses [25, 34] we presented a simple standardized exchange format for proofs and counterexample traces in software. The standardized format is still flexible enough for smaller changes and (backwards-compatible) extensions, such as our extension for multi-threaded software. This exchange format is a solution for storing analysis results in a tool-independent way with the possibility of reusing and validating them whenever a software component is touched.

Getting a test harness and an environment model for programs still requires manual effort and is an open research question. It depends on the system context, used framework, libraries and even build tools to get a well-defined harness for applying an automated verification approach. The community-driven development of a central repository of benchmark tasks for software verification [6] aims at providing standardized test and evaluation data for verification and testing tools [21, 22] to lower the burden for developers of new verification approaches. In general, it is hard to have a user-friendly software verification tool that is also precise and sound. Software

[3] http://linuxtesting.org/project/ldv
[4] https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/static-driver-verifier
[5] https://project-everest.github.io
[6] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

analyzers like Infer [7] or Frama-C [8] include support for automatically analyzing complete project structures with only minimal preparation from user-side, making it easy for the developer to get a first impression of the software under test. However, depending on the applied analysis, there is evidence that those tools lack support for correctly solving several simple tasks with specific verification properties [46, 87], e. g., the applied analysis is not completely sound and misses parts of the harness or property specification. Thus, additional effort is required for preparing such an analyzer for a project to get valid results or to compare it against other tools in research context.

To increase the usability of model checking and verification techniques in software development, the interaction with analysis tools needs to become as simple as possible. Developers and also users do not want to perform larger steps on their own or read lots of documentation, before effectively running their first verification algorithm and see the results. The entry step must be as low as possible, with the possibility to incrementally and iteratively optimize and adapt the analysis approach to the given problem.

For the developer, this can be done in several, potentially orthogonal ways. The first step is to reduce the initial burden for verifying a program, aiming towards a broader acceptance of formal methods in the area of software development. This can be done by integrating verification approaches into common build tools (like GNU Make [9], Apache Maven [10], Apache Ant [11], and many more) or integrated development environments (IDEs) (like Eclipse IDE [12], IntelliJ IDEA [13], Visual Studio Code [14]), just like it has been done for compilers, style checkers, and testing frameworks, which are already applied in development every day. The combination of testing and verification seems a beneficial goal, leveraging the best part from two worlds [72, 122], i. e., performance from testing parts of a program that are simpler to analyze in control-flow or dataflow when searching for bugs and precision from formal methods analyzing the rest of the program and generating a proof for the absence of bugs.

The next step is an automation of software verification, not only during the analysis itself, i. e., where automated software verification stands as a complementation to a manual one, but also as part of continuous integration, such that the developer is automatically informed about new bugs in his code without the need of running any verification tool locally. Such steps are already applied in larger corporations, like Google [108] or Facebook [63], as part of their commonly used tool chains, but not yet habitual for the development process in smaller companies.

The last step is the reduction of the round-trip time from starting a verification process to displaying the result towards the user. The user might want to wait a few seconds up to a few minutes after changing a smaller part of his code, and a complete check of the software should be possible overnight. There are already performance-wise comparisons of different approaches, both, different algorithms within one framework [26] as well as distinct model checking approaches

[7] https://fbinfer.com/
[8] https://frama-c.com/
[9] https://www.gnu.org/software/make
[10] https://maven.apache.org
[11] https://ant.apache.org
[12] https://www.eclipse.org/ide
[13] https://www.jetbrains.com/idea
[14] https://code.visualstudio.com

across multiple tools [16, 21]. There is evidence that caching and reusing data within a single analysis [118] and across a sequence of (incremental) verification runs is beneficial [42, 49]. Currently, the evaluation is limited to a certain kind of smaller benchmark task with up to a few thousands of lines of code. Program tasks with millions of lines of code are only partially available for analysis. For researchers, the required knowledge about an arbitrary application is simply not available, and larger companies take a deeper formal look only into specific properties or features, such as the APIs for public components [10, 59].

We are proud of CPAchecker and its related projects, which are actively applied and extended by practitioners and researchers, as part of the university studies, research, or in industry. Having further industrial partners would allow us to obtain more insights about software quality management in general, adapt or optimize the concepts, and apply our ideas as well. Overall, with the theoretical foundation in this thesis and the corresponding literature in mind, we can tackle the development and application of a user-friendly and easy-to-use, but also efficient and precise software-verification approach for the main-stream developer in the software industry.

# Acronyms

**API** application programming interface

**ARG** abstract reachability graph

**BAM** block-abstraction memoization

**BDD** binary decision diagram

**BMC** bounded model checking

**CEGAR** counterexample-guided abstraction refinement

**CPA** configurable program analysis

**CPU** central processing unit

**CTL** computation tree logic

**IDE** integrated development environment

**JNI** Java native interface

**LTL** linear temporal logic

**SMT** satisfiability modulo theories

**SMG** symbolic memory graph

# Bibliography


[1] S. B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978, https://doi.org/10.1109/TC.1978.1675141.

[2] 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, https://doi.org/10.1007/978-3-642-27940-9_4.

[3] J. Alglave, D. Kröning, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *Proc. CAV*, LNCS 8044, pages 141–157. Springer, 2013, https://doi.org/10.1007/978-3-642-39799-8_9.

[4] L. Alt, S. Asadi, H. Chockler, K. Even-Mendoza, G. Fedyukovich, A. E. J. Hyvärinen, and N. Sharygina. HiFrog: SMT-based function summarization for software verification. In *Proc. TACAS*, LNCS 10206, pages 207–213, 2017, https://doi.org/10.1007/978-3-662-54580-5_12.

[5] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state-space exploration. In *Proc. CAV*, LNCS 1254, pages 340–351. Springer, 1997, https://doi.org/10.1007/3-540-63166-6_34.

[6] P. S. Andrianov and V. S. Mutilin. Scalable thread-modular approach for data race detection. In *Proc. FISEE*, LNCS 12271, pages 371–385. Springer, 2019, https://doi.org/10.1007/978-3-030-57663-9_24.

[7] D. Baier. Integration des SMT-Solvers Boolector in das Framework JavaSMT und Evaluation mit CPAchecker. Bachelor's Thesis, LMU Munich, Software Systems Lab, 2019, https://www.sosy-lab.org/research/bsc/2019.Baier.Integration_des_SMT-Solvers_-Boolector_in_das_Framework_JavaSMT_und_Evaluation_mit_CPAchecker.pdf.

[8] D. Baier, D. Beyer, and K. Friedberger. Reproduction package (VM) for article 'JavaSMT 3: Interacting with SMT solvers in Java'. Zenodo, 2021, https://doi.org/10.5281/zenodo.4708050.

[9] D. Baier, D. Beyer, and K. Friedberger. JavaSMT 3: Interacting with SMT solvers in Java. In *Proc. CAV*, LNCS 12760, pages 195–208. Springer, 2021, https://doi.org/10.1007/978-3-030-81688-9_9.

[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, http://ieeexplore.ieee.org/document/5770931/.

[11] T. Ball and S. K. Rajamani. The Slam project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002, https://doi.org/10.1145/503272.503274.

[12] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In *Proc. SMT*, 2010, http://smtlib.cs.uiowa.edu/language.shtml.

[13] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI/ISO C specification language version 1.17, 2021, https://frama-c.com/download/acsl-1.17.pdf.

[14] T. Bergan, D. Grossman, and L. Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. In *Proc. OOPSLA*, pages 491–506. ACM, 2014, https://doi.org/10.1145/2660193.2660200.

[15] P. Berger, J.-P. Katoen, E. Ábrahám, M. T. B. Waez, and T. Rambow. Verifying auto-generated C code from Simulink - an experience report in the automotive domain. In *Proc. FM 2018*, LNCS 10951, pages 312–328. Springer, 2018, https://doi.org/10.1007/978-3-319-95582-7_18.

[16] D. Beyer. Competition on software verification (SV-COMP). In *Proc. TACAS*, LNCS 7214, pages 504–524. Springer, 2012, https://doi.org/10.1007/978-3-642-28756-5_38.

[17] D. Beyer. Software verification with validation of results (Report on SV-COMP 2017). In *Proc. TACAS*, LNCS 10206, pages 331–349. Springer, 2017, https://doi.org/10.1007/978-3-662-54580-5_20.

[18] D. Beyer. Automatic verification of C and Java programs: SV-COMP 2019. In *Proc. TACAS (3)*, LNCS 11429, pages 133–155. Springer, 2019, https://doi.org/10.1007/978-3-030-17502-3_9.

[19] D. Beyer. Advances in automatic software verification: SV-COMP 2020. In *Proc. TACAS (2)*, LNCS 12079, pages 347–367. Springer, 2020, https://doi.org/10.1007/978-3-030-45237-7_21.

[20] D. Beyer. First international competition on software testing (Test-Comp 2019). *Int. J. Softw. Tools Technol. Transf.*, 23(6):833–846, December 2021, https://doi.org/10.1007/s10009-021-00613-3.

[21] D. Beyer. Software verification: 10th comparative evaluation (SV-COMP 2021). In *Proc. TACAS (2)*, LNCS 12652, pages 401–422. Springer, 2021, https://doi.org/10.1007/978-3-030-72013-1_24.

[22] D. Beyer. Status report on software testing: Test-Comp 2021. In *Proc. FASE*, LNCS 12649, pages 341–357. Springer, 2021, https://doi.org/10.1007/978-3-030-71500-7_17.

[23] 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, https://doi.org/10.1007/978-3-540-27864-1_2.

[24] 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, https://doi.org/10.1007/978-3-319-48869-1_14.

[25] 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, https://doi.org/10.1145/2786805.2786867.

[26] D. Beyer, M. Dangl, and P. Wendler. A unifying view on SMT-based software verification. *J. Autom. Reasoning*, 60(3):299–335, 2018, https://doi.org/10.1007/s10817-017-9432-6.

[27] D. Beyer and K. Friedberger. A light-weight approach for verifying multi-threaded programs with CPAchecker. In *Proc. MEMICS*, volume 233, pages 61–71. EPTCS, 2016, https://doi.org/10.4204/EPTCS.233.6.

[28] D. Beyer and K. Friedberger. Domain-independent multi-threaded software model checking. In *Proc. ASE*, pages 634–644. ACM, 2018, https://doi.org/10.1145/3238147.3238195.

[29] D. Beyer and K. Friedberger. In-place vs. copy-on-write cegar refinement for block summarization with caching. In *Proc. ISoLA*, LNCS 11245, pages 197–215. Springer, 2018, https://doi.org/10.1007/978-3-030-03421-4_14.

[30] D. Beyer and K. Friedberger. Reproduction package for article 'Domain-independent multi-threaded software model checking' in Proc. ASE '18. Zenodo, 2018, https://doi.org/10.5281/zenodo.1322090.

[31] D. Beyer and K. Friedberger. Domain-independent interprocedural program analysis using block-abstraction memoization. In *Proc. ESEC/FSE*, pages 50–62. ACM, 2020, https://doi.org/10.1145/3368089.3409718.

[32] D. Beyer and K. Friedberger. Reproduction package for article 'Violation witnesses and result validation for multi-threaded programs'. Zenodo, 2020, https://doi.org/10.5281/zenodo.3885694.

[33] D. Beyer and K. Friedberger. Reproduction package for ESEC/FSE 2020 article 'Domain-independent interprocedural program analysis using block-abstraction memoization'. Zenodo, 2020, https://doi.org/10.5281/zenodo.4024268.

[34] D. Beyer and K. Friedberger. Violation witnesses and result validation for multi-threaded programs. In *Proc. ISoLA (1)*, LNCS 12476, pages 449–470. Springer, 2020, https://doi.org/10.1007/978-3-030-61362-4_26.

[35] 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 11. ACM, 2012, https://doi.org/10.1145/2393596.2393664.

[36] 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, https://doi.org/10.1007/978-3-540-73368-3_51.

[37] D. Beyer, T. A. Henzinger, G. Théoduloz, and D. Zufferey. Shape refinement through explicit heap analysis. In *Proc. FASE*, LNCS 6013, pages 263–277. Springer, 2010, https://doi.org/10.1007/978-3-642-12029-9_19.

[38] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011, https://doi.org/10.1007/978-3-642-22110-1_16.

[39] D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010, https://www.sosy-lab.org/research/pub/2010-FMCAD.Predicate_Abstraction_with_Adjustable-Block_Encoding.pdf.

[40] D. Beyer and T. Lemberger. Software verification: Testing vs. model checking. In *Proc. HVC*, LNCS 10629, pages 99–114. Springer, 2017, https://doi.org/10.1007/978-3-319-70389-3_7.

[41] 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, https://doi.org/10.1007/978-3-642-37057-1_11.

[42] 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, https://doi.org/10.1145/2491411.2491429.

[43] D. Beyer, S. Löwe, and P. Wendler. Refinement selection. In *Proc. SPIN*, LNCS 9232, pages 20–38. Springer, 2015, https://doi.org/10.1007/978-3-319-23404-5_3.

[44] D. Beyer and A. K. Petrenko. Linux driver verification. In *Proc. ISoLA*, LNCS 7610, pages 1–6. Springer, 2012, https://doi.org/10.1007/978-3-642-34032-1_1.

[45] D. Beyer and M. Spiessl. MetaVal: Witness validation via verification. In *Proc. CAV*, LNCS 12225, pages 165–177. Springer, 2020, https://doi.org/10.1007/978-3-030-53291-8_10.

[46] D. Beyer and M. Spiessl. The static analyzer Frama-C in SV-COMP (competition contribution). In *Proc. TACAS (2)*, LNCS 13244. Springer, 2022.

[47] D. Beyer and A. Stahlbauer. BDD-based software verification: Applications to event-condition-action systems. *Int. J. Softw. Tools Technol. Transfer*, 16(5):507–518, 2014, https://doi.org/10.1007/s10009-014-0334-1.

[48] D. Beyer and P. Wendler. Algorithms for software model checking: Predicate abstraction vs. Impact. In *Proc. FMCAD*, pages 106–113. FMCAD, 2012, ISBN: 978-1-4673-4831-7.

[49] 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, https://doi.org/10.1007/978-3-642-39176-7_1.

[50] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003, https://doi.org/10.1016/S0065-2458(03)58003-2.

[51] R. Blanc. Scala SMT-LIB. online, 2016, https://archive.softwareheritage.org/swh:1:snp:15684ff18c89d8ddb79ce78b7b33e4c8dd944916;origin=https://github.com/regb/scala-smtlib.

[52] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986, https://doi.org/10.1109/TC.1986.1676819.

[53] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *Proc. NFM*, LNCS 9058, pages 3–11. Springer, 2015, https://doi.org/10.1007/978-3-319-17524-9_1.

[54] F. Cassez and A. M. Sloane. ScalaSMT: Satisfiability modulo theory in Scala (tool paper). In *Proc. SCALA*, pages 51–55. ACM, 2017, https://doi.org/10.1145/3136000.3136004.

[55] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV*, LNCS 1855, pages 154–169. Springer, 2000, https://doi.org/10.1007/10722167_15.

[56] 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, https://doi.org/10.1145/876638.876643.

[57] E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proc. DAC*, pages 427–432. ACM, 1995, https://doi.org/10.1145/217474.217565.

[58] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999, ISBN: 978-0-262-03270-4.

[59] B. Cook. Formal reasoning about the security of Amazon web services. In *Proc. CAV (2)*, LNCS 10981, pages 38–47. Springer, 2018, https://doi.org/10.1007/978-3-319-96145-3_3.

[60] B. Cook, K. Khazem, D. Kröning, S. Tasiran, M. Tautschnig, and M. R. Tuttle. Model checking boot code from AWS data centers. In *Proc. CAV*, LNCS 10982, pages 467–486. Springer, 2018, https://doi.org/10.1007/978-3-319-96142-2_28.

[61] L. C. Cordeiro and B. Fischer. Verifying multi-threaded software using SMT-based context-bounded model checking. In *Proc. ICSE*, pages 331–340. ACM, 2011, https://doi.org/10.1145/1985793.1985839.

[62] D. Delmas and J. Souyris. Astrée: From research to industry. In *Proc. SAS*, LNCS 4634, pages 437–451. Springer, 2007, https://doi.org/10.1007/978-3-540-74061-2_27.

[63] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn. Scaling static analyses at Facebook. *Commun. ACM*, 62(8):62–70, 2019, https://doi.org/10.1145/3338112.

[64] K. Dudka, P. Peringer, and T. Vojnar. Byte-precise verification of low-level list manipulation. In *Proc. SAS*, LNCS 7935, pages 215–237. Springer, 2013, https://doi.org/10.1007/978-3-642-38856-9_13.

[65] T. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Sci. Comput. Program.*, 2(3):155–173, 1982, https://doi.org/10.1016/0167-6423(83)90013-8.

[66] B. Fischer, O. Inverso, and G. Parlato. CSeq: A sequentialization tool for C (competition contribution). In *Proc. TACAS*, LNCS 7795, pages 616–618. Springer, 2013, https://doi.org/10.1007/978-3-642-36742-7_46.

[67] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. POPL*, pages 110–121. ACM, 2005, https://doi.org/10.1145/1040305.1040315.

[68] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Proc. SPIN*, LNCS 2648, pages 213–224. Springer, 2003, https://doi.org/10.1007/3-540-44829-2_14.

[69] R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, pages 19–32. AMS, 1967, https://doi.org/10.1007/978-94-011-1793-7_4.

[70] M. Gario and A. Micheli. PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In *Proc. SMT*, 2015, https://archive.softwareheritage.org/swh:1:snp:cda89e1178b881f2d4e57978b791f5d2d7f37fd5;origin=https://github.com/pysmt/pysmt.

[71] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* LNCS 1032. Springer, 1996, https://doi.org/10.1007/3-540-60761-7.

[72] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: A new algorithm for property checking. In *Proc. FSE*, pages 117–127. ACM, 2006, https://doi.org/10.1145/1181775.1181790.

[73] Á. Hajdu and Z. Micskei. Efficient strategies for CEGAR-based model checking. *J. Autom. Reasoning*, 64(6):1051–1091, 2020, https://doi.org/10.1007/s10817-019-09535-x.

[74] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Int. J. Softw. Tools Technol. Transfer*, 2(4):366–381, 2000, https://doi.org/10.1007/s100090050043.

[75] M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *Proc. POPL*, pages 471–482. ACM, 2010, https://doi.org/10.1145/1706299.1706353.

[76] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proc. CAV*, LNCS 2725, pages 262–274. Springer, 2003, https://doi.org/10.1007/978-3-540-45069-6_27.

[77] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido. Extreme model checking. In *Verification: Theory and Practice*, pages 332–358. Springer, 2003, https://doi.org/10.1007/978-3-540-39910-0_16.

[78] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969, https://doi.org/10.1145/363235.363259.

[79] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997, https://doi.org/10.1109/32.588521.

[80] M. Huth and M. D. Ryan. *Logic in computer science - modelling and reasoning about systems (2. Edition)*. Cambridge University Press, 2004.

[81] H. Ibrhim, S. Khattab, K. Elsayed, A. Badr, and E. Nabil. A formal methods-based rule verification framework for end-user programming in campus building automation systems. *Building and Environment*, 181:106983, 2020, https://doi.org/10.1016/j.buildenv.2020.106983.

[82] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In *Proc. CAV*, LNCS 8559, pages 585–602. Springer, 2014, https://doi.org/10.1007/978-3-319-08867-9_39.

[83] X. Jia, C. Ghezzi, and S. Ying. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proc. ISSTA*, pages 177–187. ACM, 2015, https://doi.org/10.1145/2771783.2771806.

[84] R. Joshaghani, S. Black, E. Sherman, and H. Mehrpouyan. Formal specification and verification of user-centric privacy policies for ubiquitous systems. In *Proc. IDEAS*, pages 31:1–31:10. ACM, 2019, https://doi.org/10.1145/3331076.3331105.

[85] 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, https://doi.org/10.1007/978-3-319-48869-1_11.

[86] P. Kars. The application of PROMELA and SPIN in the BOS project. In *Proc. DIMACS Workshop*, volume 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 51–63. DIMACS/AMS, 1996, https://doi.org/10.1090/dimacs/032/05.

[87] M. Kettl and T. Lemberger. The static analyzer Infer in SV-COMP (competition contribution). In *Proc. TACAS (2)*, LNCS 13244. Springer, 2022.

[88] 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, https://doi.org/10.1007/978-3-642-11486-1_14.

[89] V. Kolesnykov. SMT-based model checking of concurrent programs. Bachelor's Thesis, LMU Munich, Software Systems Lab, 2020, https://www.sosy-lab.org/research/bsc/2020.Kolesnykov.SMT-based_Model_Checking_of_Concurrent_Programs.pdf.

[90] B. Kragl, C. Enea, T. A. Henzinger, S. O. Mutluergil, and S. Qadeer. Inductive sequentialization of asynchronous programs. In *Proc. PLDI*, pages 227–242. ACM, 2020, https://doi.org/10.1145/3385412.3385980.

[91] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Syst. Tech. J.*, 38(4):985–999, July 1959, https://doi.org/10.1002/j.1538-7305.1959.tb01585.x.

[92] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proc. LPAR*, LNCS 6355, pages 348–370. Springer, 2010, https://doi.org/10.1007/978-3-642-17511-4_20.

[93] M. Mann, A. Wilson, C. Tinelli, and C. W. Barrett. SMT-Switch: A solver-agnostic C++ API for SMT solving. *arXiv/CoRR*, 2007(01374), July 2020, https://arxiv.org/abs/2007.01374.

[94] K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV*, LNCS 2725, pages 1–13. Springer, 2003, https://doi.org/10.1007/978-3-540-45069-6_1.

[95] K. L. McMillan. Lazy abstraction with interpolants. In *Proc. CAV*, LNCS 4144, pages 123–136. Springer, 2006, https://doi.org/10.1007/11817963_14.

[96] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006, https://doi.org/10.1007/s10990-006-8609-1.

[97] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. Wiley, 3rd edition, 2011, ISBN: 9781118031964.

[98] P. Müller and T. Vojnar. CPAlien: Shape analyzer for CPAchecker (competition contribution). In *Proc. TACAS*, LNCS 8413, pages 395–397. Springer, 2014, https://doi.org/10.1007/978-3-642-54862-8_28.

[99] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002, https://doi.org/10.1007/3-540-45949-9.

[100] M. Nyberg, D. Gurov, C. Lidström, A. Rasmusson, and J. Westman. Formal verification in automotive industry: Enablers and obstacles. In *Proc. ISoLA*, LNCS 11247, pages 139–158. Springer, 2018, https://doi.org/10.1007/978-3-030-03427-6_14.

[101] M. Obermeier. Extending the framework JavaSMT with the SMT solver Yices2. Bachelor's Thesis, LMU Munich, Software Systems Lab, 2020, https://www.sosy-lab.org/research/bsc/2020.Obermeier.Extending_the_Framework_JavaSMT_with_the_SMT_Solver_Yices2.pdf.

[102] P. W. O'Hearn. Continuous reasoning: Scaling the impact of formal methods. In *Proc. LICS*, pages 13–25. ACM, 2018, https://doi.org/10.1145/3209108.3209109.

[103] S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *Proc. PLDI*, pages 14–24. ACM, 2004, https://doi.org/10.1145/996841.996845.

[104] 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, https://doi.org/10.1007/3-540-11494-7_22.

[105] A. Ried. Design and implementation of a cluster-based approach for software verification. Bachelor's Thesis, LMU Munich, Software Systems Lab, 2020, https://www.sosy-lab.org/research/bsc/2020.Ried.Design_and_Implementation_of_a_Cluster_Based_Approach_-for_Software_Verification.pdf.

[106] H. Riener, F. Haedicke, S. Frehse, M. Soeken, D. Große, R. Drechsler, and G. Fey. metaSMT: Focus on your application and not on solver integration. *Int. J. Softw. Tools Technol. Transf.*, 19(5):605–621, 2017, https://doi.org/10.1007/s10009-016-0426-1.

[107] B. Rothenberg, D. Dietsch, and M. Heizmann. Incremental verification using trace abstraction. In *Proc. SAS*, LNCS 11002, pages 364–382. Springer, 2018, https://doi.org/10.1007/978-3-319-99725-4_22.

[108] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan. Lessons from building static analysis tools at Google. *Commun. ACM*, 61(4):58–66, 2018, https://doi.org/10.1145/3188720.

[109] O. Sery, G. Fedyukovich, and N. Sharygina. Incremental upgrade checking by means of interpolation-based function summaries. In *Proc. FMCAD*, pages 114–121. FMCAD Inc., 2012, http://ieeexplore.ieee.org/document/6462563/.

[110] J. Sprey, C. Sundermann, S. Krieter, M. Nieke, J. Mauro, T. Thüm, and I. Schaefer. SMT-based variability analyses in FeatureIDE. In *Proc. VaMoS*, pages 6:1–6:9. ACM, 2020, https://doi.org/10.1145/3377024.3377036.

[111] C. Tian, Z. Duan, and Z. Duan. Making CEGAR more efficient in software model checking. *IEEE Trans. Softw. Eng.*, 40(12):1206–1223, 2014, https://doi.org/10.1109/TSE.2014.2357442.

[112] I. v. Langevelde, J. Romijn, and N. Goga. Founding FireWire bridges through PROMELA prototyping. In *Proc. IPDPS*, page 239. IEEE, 2003, https://doi.org/10.1109/IPDPS.2003.1213434.

[113] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency - Structure versus Automata (Proc. Banff'95)*, LNCS 1043, pages 238–266. Springer, 1996.

[114] 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, https://doi.org/10.1145/2393596.2393665.

[115] 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, http://syrcose.ispras.ru/2017/SYRCoSE2017_Proceedings.pdf.

[116] B. Wachter, D. Kröning, and J. Ouaknine. Verifying multi-threaded software with Impact. In *Proc. FMCAD*, pages 210–217. IEEE, 2013, http://ieeexplore.ieee.org/document/6679412/.

[117] L. Westhofen, P. Berger, and J.-P. Katoen. Benchmarking software model checkers on automotive code. In *Proc. NFM*, LNCS 12229, pages 133–150. Springer, 2020, https://doi.org/10.1007/978-3-030-55754-6_8.

[118] D. Wonisch and H. Wehrheim. Predicate analysis with block-abstraction memoization. In *Proc. ICFEM*, LNCS 7635, pages 332–347. Springer, 2012, https://doi.org/10.1007/978-3-642-34281-3_24.

[119] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *Proc. ISSTA*, pages 144–154. ACM, 2012, https://doi.org/10.1145/2338965.2336771.

[120] G. Yang, R. Qiu, S. Khurshid, C. S. Pasareanu, and J. Wen. A synergistic approach to improving symbolic execution using test ranges. *Innov. Syst. Softw. Eng.*, 15(3-4):325–342, 2019, https://doi.org/10.1007/s11334-019-00331-9.

[121] L. Yin, W. Dong, W. Liu, and J. Wang. Scheduling constraint based abstraction refinement for multi-threaded program verification. *CoRR*, abs/1708.08323, 2017, http://arxiv.org/abs/1708.08323.

[122] Z. Zhu, L. Jiao, and X. Xu. Combining search-based testing and dynamic symbolic execution by evolvability metric. In *Proc. ICSME*, pages 59–68. IEEE, 2018, https://doi.org/10.1109/ICSME.2018.00015.

# A. Manuscripts

This chapter provides the published research manuscripts as discussed in Chapter 2. The following manuscripts are ordered corresponding to the objectives and tasks from Chapter 1, i. e., the manuscripts are ordered and grouped semantically rather than chronologically.

Please note that the order of authors follows alphabetic scheme in most cases. The following pages contain the original manuscripts as provided by the publisher, including their own page titles, page numbers and references.


# Domain-Independent Multi-threaded Software Model Checking

Dirk Beyer
LMU Munich
Germany

Karlheinz Friedberger
LMU Munich
Germany

## ABSTRACT

Recent development of software aims at massively parallel execution, because of the trend to increase the number of processing units per CPU socket. But many approaches for program analysis are not designed to benefit from a multi-threaded execution and lack support to utilize multi-core computers. Rewriting existing algorithms is difficult and error-prone, and the design of new parallel algorithms also has limitations. An orthogonal problem is the granularity: computing each successor state in parallel seems too fine-grained, so the open question is to find the right structural level for parallel execution. We propose an elegant solution to these problems: Block summaries should be computed in parallel. Many successful approaches to software verification are based on summaries of control-flow blocks, large blocks, or function bodies. Block-abstraction memoization is a successful domain-independent approach for summary-based program analysis. We redesigned the verification approach of block-abstraction memoization starting from its original recursive definition, such that it can run in a parallel manner for utilizing the available computation resources without losing its advantages of being independent from a certain abstract domain. We present an implementation of our new approach for multi-core shared-memory machines. The experimental evaluation shows that our summary-based approach has no significant overhead compared to the existing sequential approach and that it has a significant speedup when using multi-threading.

## CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; • **Theory of computation** → *Parallel algorithms*;

## KEYWORDS

Program Analysis, Software Verification, Parallel Algorithm, Multi-threading, Block-Abstraction Memoization

**ACM Reference Format:**
Dirk Beyer and Karlheinz Friedberger. 2018. Domain-Independent Multi-threaded Software Model Checking. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238195


*ASE '18, September 3–7, 2018, Montpellier, France*

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238195

## 1 INTRODUCTION

Program verification has been applied successfully to find errors in applications or to prove their correctness. Recent hardware development aims towards parallel execution of programs either on multi-core machines or shared across several machines in a computing cluster. For large-scale program verification, we do not only need efficient algorithms, but also make use of available hardware resources up to their limits. There are some approaches to leverage such systems, but most recent algorithms for program verification and model checking are not designed to work in parallel manner and utilize only a small part of available resources. There are several reasons for this: Either the verification algorithms have dependencies between intermediate results, such that only a sequential execution is useful, or the amount of parallelism is bound by a small number, e.g., only two analyses are executed in parallel and communicate information, effectively using only a small number of CPU cores. The main question is whether and how we can (re-)design existing verification techniques such that they can be executed on parallel computer architectures.

We contribute the idea to use summaries as the objects to compute in parallel, instead of inventing new parallel state-space iteration algorithms. Block-abstraction memoization (BAM) [31] is a particularly nice method to summarize blocks of program statements, because it is independent from a particular analysis — it wraps an existing analysis without much interference and stores block summaries in a cache. We use this concept to develop a domain-independent analysis that distributes a verification problem across multiple processing units without changes to the analysis technique. Our analysis is based on a standard state-space exploration using a control-flow automaton that represents the program. The approach is orthogonal to other data-flow-based analyses, and thus, it can be combined with analyses based on different abstract domains like BDDs, explicit values, intervals, or predicates.

The value of our approach is its level of *separation of concerns*: it separates the concern of making an analysis multi-threaded from the concern of designing and implementing an abstract domain and its operators. We base our approach on BAM and use most of its data structures, such that most parts of the (wrapped) analysis (and its implementation) remain unchanged. We redesigned the algorithm such that we can efficiently execute it across several processing units. The parallelism of the analysis is only bound by the structure of the program to be analyzed and the amount of work found during the analysis. Our work includes a transformation of the existing algorithm of BAM from a sequential, recursively defined algorithm into a parallel approach. Additionally, we benefit from the existing infrastructure of BAM, i.e., we also use a cache for block abstractions and apply the operators reduce and expand to increase the cache hit rate. The analysis is sound, implemented

in the open-source verification framework CPAchecker, and can be combined with existing components of the framework, including CEGAR [20] or witness export [5, 6].

**Contributions.** We make the following contributions:

- We introduce a new technique for parallelization of verification algorithms that is independent from particular abstract domains because it is based on a flexible and configurable block summarization.
- We implemented the technique in the open-source verification framework CPAchecker. Our implementation and all experimental data are available to other researchers and practitioners for replication via our artifact [9] and supplementary website.[1]
- We evaluated our new technique on a large set of benchmarks and show (1) that the parallel version of BAM (if using only one CPU core) behaves similar to the sequential version (i.e., there is no significant overhead for parallelization) and (2) that the parallel version of BAM significantly improves the response time of the verification process for programs that are large enough to benefit from multi-threading.

**Related Work.** The idea to use parallel algorithms in software verification is not new. There exist several approaches reaching from plain parallel execution of different algorithms (until the first analysis succeeds) via one-way communication between (some) analyses (one analysis provides additional information for another one) to fully parallel analyses (dividing the state space into partitions that are explored separately).

*Portfolio Approaches.* A simple, but effective approach is to run a portfolio analysis [24], i.e., a fixed number of predefined analyses in parallel to leverage the available CPU cores on a single machine, such that the verifier terminates with the first succeeding analysis (e.g. [22, 26]). This strategy is applied either to separately explore the state space, e.g., with different domains, or in a way that one analysis provides information for another one, for example to enrich it with additional invariants [7]. Such approaches for parallel software verification are not scalable due to its fixed number of different analyses, and they suffer from the problem that each single analysis only uses a small fraction of the available resources. If all but one analysis fail to determine a verification result (because of unsupported features in the task, or imprecision of the analysis), the remaining work is sometimes limited to a single analysis and thus a single core.

*Multi-Threading Approaches.* SPIN [23] and Divine [3, 29] are based on pure explicit model checking and use a central hash table to check for existing (already analyzed) states. LTSmin [18] either performs explicit state-space search in a parallel manner or uses a BDD-based approach using the BDD-library Sylvan [30] that internally parallelizes its operations. Other approaches divide a given problem into smaller components that are verified separately, before joining the results to get a proof for a whole program [21, 25]. An example implementation for such a technique is the tool SoftVer that uses BDDs and predicates.

Structurally-defined conditional analysis [28] is an approach that splits a program according to conditions as in conditional model checking [11], that is, given a program $P$ and a condition $\psi$, two analysis instances can be created, one conditional analysis of $P$ and $\psi$ and one conditional analysis of $P$ and $\neg\psi$. The two analysis instances are completely independent and can be executed in parallel. The approach can scale up to an arbitrary number of splits. The elegance of this approach is that it does not depend on a specific implementation but can be built on top of existing, off-the-shelf tool components.

*Multi-Machine Approaches.* State-space exploration can be distributed across several machines by partitioning the possible state-space. Tools like SPIN [23], CSeq-Swarm [27], or the SPARK Analysis Tools [19] divide the verification problem after a short pre-analysis of the program, and split the potential state space and the verification condition according to given time and memory limitations, available processing units, or other criteria. This approach is potentially problematic due to the unknown nature of the program to be analyzed, e.g., it might not match the pre-defined scheduling. For degenerated state spaces, the parallel analysis might be imbalanced between different threads/processes. Other tools like LTSmin [18] or Divine [3, 29] circumvent such imbalances by a dynamic scheduling approach. The approach of structurally-defined conditional analysis [28] can also be extended to benefit from multi-machine environments.

Our contribution is a more general parallel technique for program analysis and can be applied to an arbitrary domain and even combinations of several domains. Thus, explicit-value analysis, BDD-based analysis, as well as predicate analysis can benefit from our approach. The parallelism of the approach presented in this paper is based on the internal structure of the program, i.e., an automatic partitioning of the control flow, and tries to use all available processing units, only depending on the dynamic behavior of the program analysis, i.e., the unfolding of the abstract state space.

## 2 BACKGROUND

The following section provides an overview of basic concepts and definitions that our approach is based on. We describe the program representation, configurable program analysis, the details of block-abstraction memoization, and how we advanced it towards an efficient parallel algorithm for program analysis (for more detail see the original articles [12, 31]).

### 2.1 Program Representation

We restrict the presentation to a simple imperative programming language, where all operations are either assignment or assume operations. A program is represented by a *control-flow automaton* (CFA) $A = (L, l_0, G)$, which is a directed graph consisting of a set $L$ of program locations (modeling the program counter), a set $G \subseteq L \times Ops \times L$ of control-flow edges (modeling the computation steps from one location to the next: assignment or assume operations), and an initial program location $l_0$ (entry point of the program).

### 2.2 CPA and CPA Algorithm

A *configurable program analysis* (CPA) [12] is specified by an abstract domain for a program analysis and operators to model the behavior of the program analysis: A CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$ consists of

[1] https://www.sosy-lab.org/research/bam-parallel/

(1) an abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ that consists of a set $C$ of concrete states, a lattice $\mathcal{E} = (E, \sqsubseteq)$ over a set $E$ of abstract-domain elements (i.e., abstract states) and a partial order $\sqsubseteq$, and a concretization function $[\![\cdot]\!]$ that maps each abstract-domain element to the represented set of concrete states.
(2) a transfer relation $\rightsquigarrow \subseteq E \times E$ that yields successors of an abstract state.
(3) a merge operator merge $\subseteq E \times E \to E$ that determines how to merge two abstract states when control flow meets).
(4) a termination check stop $\subseteq E \times 2^E \to \mathbb{B}$ that specifies whether an abstract state is covered by a set of abstract states.

Algorithm 1 CPAalg performs a state-space exploration. It computes an overapproximation of the reachable states by constructing abstract states for the program based on a given CPA and an initial abstract state. The algorithm is a fixed-point iteration and maintains a set waitlist of abstract states that still have to be explored, and a set reached of already explored abstract states. In each iteration, the algorithm takes an abstract state from waitlist (line 2) and computes its successors (line 3). The algorithm checks whether a new state can be merged with an existing state, and updates the work sets accordingly (lines 5–8). The operator stop ensures that the new abstract state is only added to the work sets if the abstract state is not already covered by any of the existing states in reached (lines 9–11). The algorithm terminates if either the set waitlist is empty or there is another reason to abort early, e. g., a property violation.

We use a simplified version of algorithm CPAalg [8] in order to shorten the presentation. The precision and precision adjustment, which determine the granularity of the analysis within a CEGAR loop, are neglected in this description, but fully available and supported in our implementation.

Different aspects of a program are analyzed by different CPAs, and compositions of CPAs allow more advanced analyses. CPAs have been defined for many abstract domains, such as BDD-based analysis [17], (explicit or symbolic) value analysis [14, 15], predicate analysis [8, 10, 13], or combination thereof [2]. Also the tracking of the program counter and of the call stack for procedures are defined as CPAs. We will not go into detail for all their definitions and descriptions here, because our approach works on an abstract level and is independent from a specific domain. For our evaluation later, we use a value analysis that tracks variables and their values explicitly, e.g., an abstract state is a (partial) function that maps program variables to values.

**Algorithm 1** CPAalg($\mathbb{D}$, reached, waitlist), taken from [8]

**Input:** a CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$,
where $E$ denotes the set of elements of the lattice of $D$,
a set reached $\subseteq E$ of abstract states,
a set waitlist $\subseteq$ reached of frontier abstract states,
a function abort : $E \to \mathbb{B}$ that defines whether the algorithm should abort early
**Output:** the updated sets reached and waitlist

```
1:  while waitlist ≠ ∅ do
2:    pop(e) from waitlist
3:    for each e' with e ⇝ e' do
4:      for all e'' ∈ reached do
5:        e_new := merge(e', e'')
6:        if e_new ≠ e'' then
7:          reached := reached ∪ {e_new} \ {e''}
8:          waitlist := waitlist ∪ {e_new} \ {e''}
9:      if ¬stop(e', reached) then
10:       reached := reached ∪ {e'}
11:       waitlist := waitlist ∪ {e'}
12:       if abort(e') then
13:         return (reached, waitlist)
14: return (reached, waitlist)
```

## 2.3 BAM

Block-abstraction memoization (BAM) [31] is a modular approach for reachability analysis of abstract state graphs (such as abstract models of programs). Therefore, it treats a large program as a set of *blocks*, and analyzes the blocks separately. The result of a block analysis (the *block abstraction*) of a nested block is embedded in the surrounding block's analysis. Block abstractions are also stored in a cache for later reuse in order to avoid repeated computation of the same block abstraction, to speed up the analysis. BAM defines the two operators reduce and expand that aim at a higher cache hit rate. For simplicity we will neglect both operators in the further description. They are orthogonal to the approach of parallel analysis that we present here.

The components of BAM are defined in detail in the following:

*2.3.1 Blocks.* The basic components of BAM are *blocks*, which are formally defined as parts of a program: A block $B = (L', G')$ of a CFA $A = (L, l_0, G)$ consists of a set $L' \subseteq L$ of connected program locations and a set $G' = \{(l_1, op, l_2) \in G \mid l_1, l_2 \in L'\}$ of control-flow edges. Two different blocks $B_1 = (L'_1, G'_1)$ and $B_2 = (L'_2, G'_2)$ are either disjoint ($L'_1 \cap L'_2 = \emptyset$) or one block is completely nested in the other block ($L'_1 \subset L'_2$). Each block $B = (L', G')$ has *entry* and *exit locations*, which are defined as $In(B) = \{l \in L' \mid (\exists (l', op, l) \in G \wedge l' \notin L') \vee \nexists (l', op, l) \in G\}$ and $Out(B) = \{l \in L' \mid (\exists (l, op, l') \in G \wedge l' \notin L') \vee \nexists (l, op, l') \in G\}$, respectively. In general, the block size can be freely chosen in BAM. In most cases, functions and loops are used as block size, because they represent the logical structure of a program and lead to natural block abstractions.

Figure 1 shows a schematic example of a CFA and how it could be divided into blocks. It does not show any operations; we omit details for ease of presentation. The largest block (denoted as $B_A$) consists of all locations and represents the whole CFA of the program. The other blocks (denoted as $B_B$ to $B_F$) are smaller and consists of fewer locations. Block $B_F$ is nested in block $B_E$, which in turn is nested in block $B_A$. Location 3 is the entry location of block $B_B$, i.e., $In(B_B) = \{3\}$, and location 4 is its exit location, i.e., $Out(B_B) = \{4\}$.

*2.3.2 BAM-CPA.* The basis of CPAchecker is the idea of configurable program analysis. Thus, BAM is formalized as a CPA $\mathbb{BAM} = (D_{\mathbb{BAM}}, \rightsquigarrow_{\mathbb{BAM}}, \text{merge}_{\mathbb{BAM}}, \text{stop}_{\mathbb{BAM}})$. BAM works on an abstract, domain-independent level and uses an abstract-domain-dependent wrapped analysis (like the BDD-based, explicit value, interval, or predicate analysis) to track variables and values. This wrapped analysis is also given as CPA $\mathbb{W} = (D_{\mathbb{W}}, \rightsquigarrow_{\mathbb{W}}, \text{merge}_{\mathbb{W}}, \text{stop}_{\mathbb{W}})$, based on which we now formalize BAM:

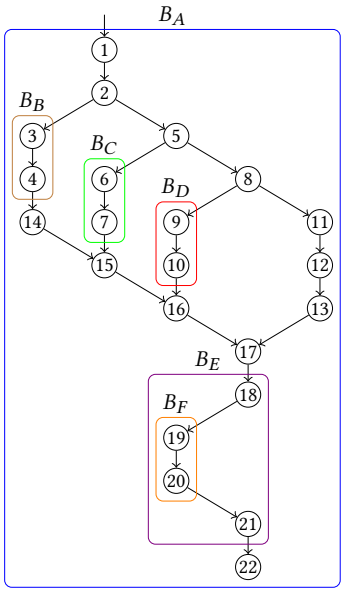



**Figure 1: Schematic control-flow automaton with blocks**

(1) The domain $D_{\mathbb{BAM}}$ wraps the domain $D_{\mathbb{W}}$.
(2) The transfer relation $\rightsquigarrow_{\mathbb{BAM}}$ for a block $B$ has a transfer $s \rightsquigarrow_{\mathbb{BAM}} s'$ for two abstract states $s$ and $s'$ if

$$s' \in \begin{cases} \{s'' \mid s \overset{B_{sub}}{\rightsquigarrow}_{\mathbb{BAM}} s''\} & \text{if } l \in In(B_{sub}) \text{ // apply BAM to } B_{sub} \\ \{s'' \mid s \rightsquigarrow_{\mathbb{W}} s''\} & \text{if } l \notin Out(B) \text{ // delegate to } \mathbb{W} \end{cases}$$

where $l$ is the program location of $s$.
Depending on the currently analyzed program location $l$, the transfer relation chooses between two possible steps: For an entry location of a block $B_{sub}$, the operation $\overset{B_{sub}}{\rightsquigarrow}_{\mathbb{BAM}}$ represents the block abstraction for the block $B_{sub}$ and the block-entry abstract state $s$. The block abstraction is computed by a call CPAalg($D_{\mathbb{BAM}}$, $\{s\}$, $\{s\}$). For exit locations of blocks, there is no succeeding abstract state (in the analysis of the current block $B$). For other program locations, the wrapped transfer relation $\rightsquigarrow_{\mathbb{W}}$ is applied.
(3) The merge operator $\text{merge}_{\mathbb{BAM}} = \text{merge}_{\mathbb{W}}$ and the termination check $\text{stop}_{\mathbb{BAM}} = \text{stop}_{\mathbb{W}}$ correspond to the wrapped analysis.

The performance of BAM can easily be increased by a cache $\text{cache} \subseteq (Blocks \times E) \rightarrow (2^E \times 2^E)$, which maps a block and an entry abstract state of the block to the set of reached abstract states and the set of frontier states. The cache is optional for the application of BAM, but the memoization of block abstractions improves the performance. Additionally, the operators reduce and expand can be applied for a higher cache hit rate. We ignore them for simplicity.

## 2.4 Towards Parallel BAM

A simple state-space exploration that enumerates all reachable abstract states and only checks whether they were already part of the set reached can be done with the operators $\text{merge}_{sep}$ and $\text{stop}_{sep}$ (defined as $\text{merge}_{sep}(e, e') := e$ and $\text{stop}_{sep}(e, R) := \exists e' \in R : e \sqsubseteq e'$, or even with a simpler form $\text{stop}_{sep}(e, R) := \exists e' \in R : e = e'$). Well-known techniques for explicit-state model checking [3, 23] use such an approach to analyze the state space. This approach can be parallelized easily by synchronizing the access to the existing abstract states in the sets reached and waitlist and applying the operators $\rightsquigarrow$, $\text{merge}_{sep}$, and $\text{stop}_{sep}$ concurrently. With lock-free implementations of the set data structures for reached and waitlist there is only minimal synchronization necessary for an efficient analysis. However, when using more general (and possibly more expensive) operator instances, the complete sets reached and waitlist (and also larger parts of the CPA algorithm) would need to be locked to ensure single-thread access, which prevents an efficient parallel application of the algorithm.

To circumvent this problem, our new approach does not introduce parallelism within the CPAalg algorithm, but applies several independent CPAalg instances in parallel. Each CPAalg invocation is executed in a separate thread on its own part of the state space, i.e., with its own sets reached and waitlist of abstract states, such that there is only minimal communication between the algorithm instances. The necessary infrastructure for such an approach is based on BAM. The previously given basic definition of BAM leaves room for several implementation details, such that both (the sequential and the parallel) implementation match the given specification. The computation and application of block abstractions can be done in sequential or parallel manner.

# 3 PARALLEL BAM

Our contribution is a scalable parallelization of the sequential algorithm of BAM. Block abstractions are independent from each other and also from the surrounding context. Thus, they can be computed in parallel, as soon as the initial abstract state of a block abstraction is known. The sequential version of BAM, which was defined by Wonisch and Wehrheim [31], recursively calls another CPA algorithm for each newly entered block, waits for its termination and directly uses the result as a block abstraction of the entered block. In contrast to that, our parallel version schedules the computation of a nested block abstraction in another thread and continues with the analysis of further abstract states from the set waitlist.

Each block abstraction is computed by a separate instance of the CPAalg algorithm (in own thread), with own instances of the sets reached and waitlist, and a thread-safe instance of the transfer relation $\rightsquigarrow$ and the operators merge and stop. The operators are stateless, and thus can be used in parallel from several threads. There is no need to lock the data structures of a CPAalg instance. In parallel algorithms, a critical point is the number of synchronizations. Block abstractions are *large enough* to avoid expensive synchronization for single steps during the computation. Synchronization is only needed when entering or leaving a block, i.e., when starting or terminating a block's analysis instance. Additionally, the communication only happens between dependent block abstractions, such that no global locking is required in the algorithm.

## 3.1 Jobs as Components with Dependencies

Our technique is based on the parallel execution of components named *jobs*. A job $job = (\mathbb{D}, \text{reached}, \text{waitlist}, B)$ consists of

- a CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$ that determines the analysis (in our case we always set $\mathbb{D} = \mathbb{BAM}$),
- a set reached and a set waitlist of abstract states to be analyzed, and
- a block $B = (L', G')$ representing the partition of the program's CFA to be analyzed.

A job is executed by applying Alg. 1 CPAalg with the given CPA $\mathbb{D}$ on the sets reached and waitlist. Note that there can be several jobs for the same block $B$, but each set reached and each set waitlist are assigned to exactly one job. There are no shared data based on abstract states for different jobs. This allows us to execute jobs in parallel, because the job executions are independent from each other. If a block has nested blocks, the corresponding block abstraction depends on the block abstractions of those nested blocks. In the sequential implementation of BAM, the dependencies of block abstractions on nested-block abstractions are implicitly solved by calling algorithm CPAalg recursively, i.e., the analysis of an outer block waits until a nested block abstraction is computed completely, and then continues. In the parallel approach we explicitly maintain such dependencies between (analyses of) block abstractions. A relation deps $\in$ *jobs* $\times E \times$ *jobs* tracks at which abstract state a block abstraction needs to be computed and applied. This relation needs to be globally visible, shared across all threads, and modifications are applied atomically. As dependencies are only modified when a job is started or terminated, the overhead for synchronization is negligible. Our implementation does currently not support recursive tasks and thus there are no cyclic dependencies between block abstractions.

## 3.2 Scheduling and Job Execution

The parallel execution of analyses needs a scheduling algorithm that distributes the parallel running analyses onto the available processing units. In our case we chose a simple task queue from the Java Concurrency API, where we insert our jobs, and let the framework do the scheduling. We can set the number of running threads to the available hardware by using the default Java thread pool. For simplicity of Alg. 3, the actual scheduling is hidden in the call *schedule* that (asynchronously) executes the given job with the given data.[2] This solution has only small overhead (for run time and for developers) and is performant enough for the analysis, even when applied to a larger scale of computing resources. We have nearly linear speedup when using multiple cores (see Sect. 4), thus we assume that the build-in scheduling is efficient enough for our currently available hardware.

The basic idea of a parallel implementation is given in Algs. 2 and 3. The function abort of Alg. 1 CPAalg terminates the analysis as soon as a nested block abstraction needs to be computed. In this case, we determine the necessary data to compute the block abstraction in our scheduling algorithm and schedule a new analysis to compute the nested-block abstraction asynchronously. The abstract state before entering the block is removed from the current set waitlist and stored as a part of the dependency relation deps. After the computation of the nested-block abstraction is finished, the dependency is removed from deps and the state is re-added into the set waitlist. The function schedule executes the given job asynchronously with algorithm Alg. 3. The asynchronous execution of a job can be delayed due to limited resources or because the same job is scheduled twice, i.e., with different arguments. We use a thread pool for scheduled jobs based on a job queue with a FIFO ordering strategy. The function scheduleAndWait does the same, but awaits the termination of the job. The method registerBA is executed whenever a block analysis terminates. It extracts the block abstraction from the analyzed set reached and updates the cache of BAM.

[2]The pseudo code omits some scheduling-related code, as this would be too much detail for this description and can be looked up in our reference implementation.

**Algorithm 2** ParallelBAM($\mathbb{D}$, reached, waitlist): Initial step for parallel BAM

**Input:** a CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$,
where $E$ denotes the set of elements of the lattice of $D$,
a set reached $\subseteq E$ of abstract states,
a set waitlist $\subseteq$ reached of frontier abstract states,
a global relation deps $\subseteq$ *jobs* $\times E \times$ *jobs* to track computations of block abstractions
**Output:** a set of reachable abstract states,
a subset of frontier abstract states

```
1: deps := ∅
2: mainJob := (𝔻, reached, waitlist, mainBlock)
3: JobExecutor(mainJob, deps, ∅)
4: return (mainJob.reached, mainJob.waitlist)
```

**Algorithm 3** JobExecutor(job, deps, statesToAdd): Job execution for parallel BAM

**Input:** a job = $(\mathbb{D}, \text{reached}, \text{waitlist}, B)$,
a global relation deps $\subseteq$ *jobs* $\times E \times$ *jobs* to track computations of block abstractions,
a set statesToAdd $\subseteq E$ of abstract states to be added before starting the analysis

```
1: job.waitlist := job.waitlist ∪ statesToAdd
2: deps := deps \ {(job, e, ·) ∈ deps | e ∈ statesToAdd}
3: job.reached, job.waitlist :=
          CPAalg(𝔻, job.reached, job.waitlist)
4: missingBAs := {e ∈ reached|hasMissingBA(e)}
5: if missingBAs ≠ ∅ then    // nested BA needed
6:   for e ∈ missingBAs do
7:     job.waitlist := job.waitlist \ {e}
8:     childJob := (ℂ, {e}, {e}, getBlock(e))
9:     deps := deps ∪ {(job, e, childJob)}
10:    schedule(childJob, deps, ∅)
11:  schedule(job, deps, ∅)
12: else
13:   finished := job.waitlist = ∅ ∧ {(job, ·, ·) ∈ deps} = ∅
14:   shouldAbort := ∃e ∈ job.reached : abort(e)
15:   if finished ∨ shouldAbort then
16:     registerBA(job.reached, shouldAbort)
17:     parents := {(·, ·, job) ∈ deps}
18:     for (parentJob, updateState, ·) ∈ parents do
19:       schedule(parentJob, deps, {updateState})
20:     deps := deps \ parents
```

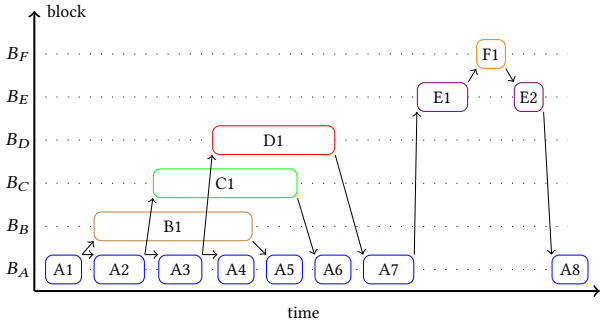



**Figure 2: Schematic time line of a possible execution of jobs with parallel BAM for the example in Fig. 1**

### 3.3 Example Application of Parallel BAM

The CFA in Fig. 1 consists of two characteristic parts: the upper part has heavy branching and several control-flow paths, the part below location 17 consists of a simple chain of locations. Parallel BAM implicitly recognizes this structure and the scheduling will apply a parallel analysis for the upper part. Figure 2 shows a possible time line for the execution of the new algorithm for the CFA given in Fig. 1. The heavy branching part of the program results in independent blocks $B_B$, $B_C$, and $B_D$, which can be analyzed in parallel. Each box in Fig. 2 represents a job, consisting of a CPA $\mathbb{W}$, a set reached, a set waitlist, and a block $B \in \{B_A, ..., B_F\}$. For each block (more concretely: for each set reached), there can be several jobs that are applied in sequential order.

For the example, let us assume a depth-first search as iteration order and an expensive computation in the blocks $B_B$, $B_C$, and $B_D$. In general, the iteration order for the program analysis can be configured by the user, and the effort to analyze blocks depends of course on the given task.

Initially, Alg 2 creates job $A1$ (Alg. 2, line 2) for the analysis of the block $B_A$. Figure 2 shows the execution of job $A1$ with Alg. 3 as a box along the time axis. Internally, Alg. 1 CPAalg analyzes the first abstract states of the given task (Alg. 3, line 3), until the entry location of block $B_B$ is reached. Algorithm CPAalg terminates for the job $A1$ and two further (independent) jobs $A2$ and $B1$ are scheduled (Alg. 2, line 10 and 11) and executed in parallel. The job $B1$ analyses the block $B_B$ and is not interrupted by another block-entry location. The job $A2$ is scheduled because there is a branching at location 2 in the CFA, such that the set waitlist of the terminated CPAalg in job $A1$ was not empty.

For the example, we assume that the job $A7$ analyzes the program location with CFA location 17. For the part below location 17 however, inter-block dependencies prevent a parallel execution of jobs and we need to explicitly wait for nested-block abstractions to be computed. In Fig. 2 this is visible for jobs $E1$, $F1$, $E2$, and $A8$, which do not have any parallel execution. Overall, our parallel version of BAM uses a dynamic scheduling, such that such imbalances are prevented in most cases.

### 3.4 Soundness of the Parallel Approach

We take a short look at the soundness of the parallel algorithm based on its sequential instance. The main difference between the sequential and the parallel version of BAM is the computation order of block abstractions. Instead of computing one block abstraction after another, they are computed in parallel whenever possible. As the computations of block abstractions themselves are independent and do not share any relevant data, the theoretical basis for soundness does not change. Thus, the parallel approach is as sound as the sequential algorithm that was proven to be sound in [31], i.e., only the iteration strategy for the state space differs and the soundness relies on the underlying analysis $\mathbb{W}$ of BAM. In other words: If there exists an abstract path in the analyzed source file that reaches a property violation, then the same path is also explored by the parallel algorithm, consisting of the same block abstractions and abstract states as computed by a sequential analysis.

### 3.5 Requirements for Parallel Execution

Our parallel approach has some additional requirements on the used components: Each used CPA has to allow multi-threaded access to its main components, the operators must be thread-safe and usable in parallel. This can either be implemented (a) by stateless operators (which is the intended behavior of operators anyway) or (b) by separate instances of the operators for each accessing thread (including independent data structures). **(a)** An ideal framework would only have stateless operators (just as their theoretically defined mathematical pendant) and thus, they would easily be usable in multi-threaded context without locking or synchronization. **(b)** While the operators are stateless in theory, a large software system (such as the framework CPAchecker), where the developers integrate several different theoretical approaches, requires an implementation that partially deviates from the concept of stateless operators. We noticed that the transfer relation $\rightsquigarrow$ and also the operators merge and stop for several CPAs were already designed and implemented in a stateless manner, such that they can easily be used for our parallel BAM implementation. Depending on the CPA, most of the code (and also most of the theoretical background) is placed in the transfer relation, and thus the conceptual difficulty was to rewrite those parts that are critical and might need to be synchronized. To avoid heavy synchronization, we have converted some (non-critical) parts like statistics and time measurement into a thread-safe implementation or provide independent instances of operators for special cases.

We have not only added the new algorithms (Alg. 2 and 3) for parallel BAM into the framework, but also modified some other components such that they can be combined and used with the new algorithm. The following list contains a few corner cases of CPAs that were touched or are usable with our approach:

- LocationCPA: The program location for the current analysis is tracked with the LocationCPA. As the program location of each statement is constant after parsing the program and written into the CFA location, the 'state' of the operators is the (immutable) CFA itself. Thus no changes had to be made.
- CallstackCPA: The call stack for the current analysis is determined by the CallstackCPA. As the corresponding operators are stateless (i.e., only depending on the abstract call-stack state given as parameter), no changes were required.
- ValueCPA: The ValueCPA performs an explicit-value analysis and tracks numerical values for variables. The analysis itself does not need to be changed for synchronization.

## 4 EVALUATION

This section compares our new parallel approach with the existing sequential implementation and shows that the new approach can reduce the response time considerably when executed on several cores. First, we compare the old sequential implementation with the new implementation (executed with only one thread), in order to show that no regression appears and that both analyses behave as similar as possible. Second, we explore the speedup of the analysis depending on the number of threads (as far as our hardware allows).

### 4.1 Evaluation Goals

It is clear from theory that not all verification tasks will benefit from our parallelized verification approach: (a) There are many programs where most paths have sequential dependencies between blocks and therefore, there is not much room for performance improvements from parallelization, and (b) there are many small programs, for which parallelization does not make a difference. We claim that our approach is effective in both regards: it parallelizes and speeds up verification process (response time) *if* the structure of the program contains sufficient branching and the size of the program is large enough *and* does not negatively influence the performance for those verification tasks that are small or have sequential dependencies.

**Claim 1.** The BAM-based approach to parallelization does not negatively impact the performance of verification tasks overall. *Evaluation Plan:* We take a large benchmark set of verification tasks and verify them with and without parallelization, restricted to one processing unit. If the run time is not worse for the parallel version, then the claim is valid.

**Claim 2.** The BAM-based approach to parallelization reduces the response time of verification tasks by leveraging several processing units. *Evaluation Plan:* We take a large set of verification tasks that can potentially benefit from parallelization and compare the response time of the verification with different numbers of processing units.

If this experiment is positive, the question raises where the benefit comes from: Is it the BAM-based approach to parallelization, or are there other technical components of the verifier that contribute to the speed up? What are the configurable parts of the verifier that can benefit from parallelization? Can they be controlled in an experiment (switched on and off separately)?

**Claim 3.** The parallelization of the program analysis using BAM contributes considerably to the speedup. *Evaluation Plan:* After identifying variables to control, we run experiments to investigate the influence of the identified components.

### 4.2 Benchmark Environment and Limitations

*Benchmark Sets.* For our evaluation we use a large subset of the SV-Benchmarks repository [4] containing over 5 400 verification tasks[3], sorted into different categories according their specification, internal structure, or behavior. For the comparison of the existing sequential implementation with the new parallel approach (limited to one CPU core), we use all verification tasks with a reachability property, in order to evaluate on a diverse set that the approach has no negative effect (Claim 1). To demonstrate the positive effect of parallelization of the new approach, we chose those verification tasks from the category *ReachSafety-ECA* that consists of rather large problems with a highly branching control flow (Claim 2).

*Setup.* We ran the experiments on a cluster of 168 identical machines with a hardware specification that roughly matches available resources on machines of software developers. This way, replication of our experiments does not require specific hardware. For each single verification run we limit the CPU time to 15 min and the memory to 15 GB, and we use an Intel Xeon E3-1230 v5 CPU with 3.40 GHz with 8 processing units (4 physical cores with hyper-threading). The limit of CPU time enables us to even compare the effectiveness of parallelization (response time vs. CPU time) for those verification tasks for which the verifier runs into a timeout. We evaluated our implementation in CPAchecker[4], revision `r28809`, from the official project repository[5].

Because we use Intel processors with hyper-threading, where two neighboring (virtual) processing units share some hardware components and influence each other, we pair the (virtual) processing units and use a step width of 2 for our experiments with varying number of processing units, i.e., we use 2, 4, 6, 8 processing units and omit the odd numbers of processing units. The benchmarking framework BenchExec [16] takes care of correctly assigning the two processing units of the same physical core together to the verification processes. We report all times in seconds and use the term *CPU time* for the accumulated usage of processing units of a CPU, and the terms *response time* or *wall time* for the time that elapses between the start and the termination of the verification run.

*Analysis Configuration.* We configure BAM to use function and loop bodies as blocks. BAM can be combined with several analyses; for our evaluation, we choose a combination where the performance influence from additional components is small: BAM with an explicit-value analysis (VA) without CEGAR. This way, we configure a simple state-space exploration based on an explicit tracking of variables and their values. Both the sequential and the parallel configurations apply a depth-first-search as exploration strategy, i.e., the set waitlist of the CPA algorithm is a FIFO queue for each configuration.

Unfortunately, we can not compare to other multi-threading verifiers for reachability properties of sequential C programs, because there exists no equivalent approach to the best of our knowledge (cf. related work in the introduction; there are portfolio verifiers).

### 4.3 Claim I: Sequential vs. Parallel Algorithm

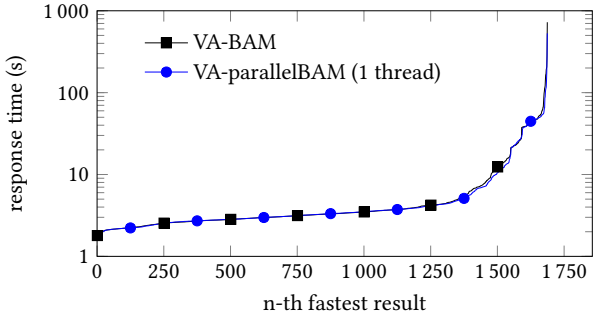
*Configuration.* In our first experiment we compare the existing sequential algorithm with the new parallel algorithm. Therefore, we run all experiments on only one processing unit. The FIFO ordering of the job queue (see Sect. 3.2) in the parallel algorithm guarantees that block abstractions are computed in the same order as their blocks are reached, i.e., it behaves as similar as possible to the sequential algorithm.

*Results.* Figures 3a and 3b show the response time of the configurations for the benchmark set containing all verification tasks with a reachability property. A quantile plot contains graphs that indicate the quantile of solved problem instances (x-axis) each

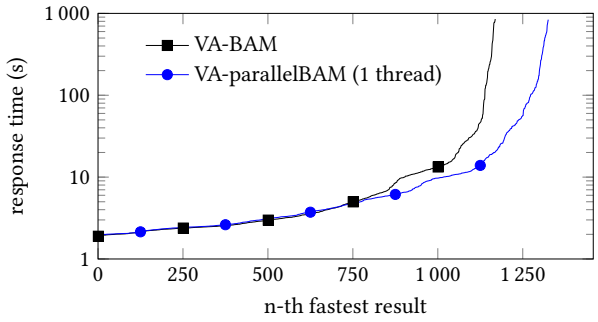
[3] https://github.com/sosy-lab/sv-benchmarks

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

[5] https://gitlab.com/sosy-lab/software/cpachecker




**(a) Verification tasks with correctness proof**




**(b) Verification tasks with property violation**

**Figure 3: Quantile plots for results of BAM with value analysis, sequential compared to parallel version with one thread**

within a certain response time (y-axis).[6] It does not show a direct comparison for individual verification tasks, but allows to compare the overall behavior of an analysis configuration. We divided the benchmarks into two groups: The plot in Fig. 3a contains results for all verification tasks for which a correctness proof was computed; the plot in Fig. 3b contains results for all verification tasks for which a property violation was found. The overall impression is that the (single-threaded) parallel technique does not have any noticeable overhead above the sequential approach, i.e., the scheduler and the job executor from Alg. 3 are efficient. The new approach behaves almost identical when computing proofs, and for finding property violations, it is even faster and can solve more problems, which we discuss in the following.

*Discussion.* The difference in Fig. 3b between the verification approaches results from the exploration order of the state space. After a nested-block abstraction has been computed, there is a small difference in the sorting of abstract states in the sets waitlist of both approaches. The existing sequential analysis has (and keeps) the abstract states in the set waitlist. The (single-threaded) parallel approach removes abstract states when finding a missing block abstraction (cf. Alg. 3, line 7) and re-adds those abstract states into each set waitlist (cf. Alg. 3, line 1) after computing the necessary block abstraction. There are small differences in the exploration order and depending on the task's structure, different paths might be analyzed first. In those cases, the parallel approach does not apply a pure depth-first exploration order, but partially prefers paths that do not traverse deeply nested blocks, which seems beneficial when it comes to finding property violations. For this reasoning, we conclude that for evaluating Claim II, it would not be valid to consider the verification tasks with property violations, because the variable "exploration order" is not controlled.

We conclude that Claim 1 holds, because we did not observe any negative impact of our new approach.

## 4.4 Claim II: Scalability of Parallel BAM

*Configuration.* We show the effectiveness of the parallelization of our new approach by increasing the number of threads (2, 4, 6, 8 threads) and observe the improvement of the response time. The upper limit of the number of threads is determined by the hardware that we use. We set the number of processing units assigned to the verification process to be equal to the number of threads. We chose a subset of 154 tasks from the category *ReachSafety-ECA*, such that they need a reasonable amount of time (at least 3 s with only one thread) and do not contain a property violation. With a too small analysis time, the default overhead of the CPAchecker framework itself (like JVM startup time or parsing time) hides the effect of the parallel approach and blurs the picture. Additionally, finding a path to a property violation with a parallel verification approach easily leads to non-deterministic results if there are several property violations in a verification task or a property can be reached via different program paths[7]. Thus, we select from the benchmark set only those verification tasks without property violation, in order to make sure to compare the response time that is necessary to analyze the whole state space. The used benchmark set consists of three groups: 47 simple tasks, 36 medium tasks, and 71 difficult tasks. The difficulty is roughly given by the size of the state space to be explored.

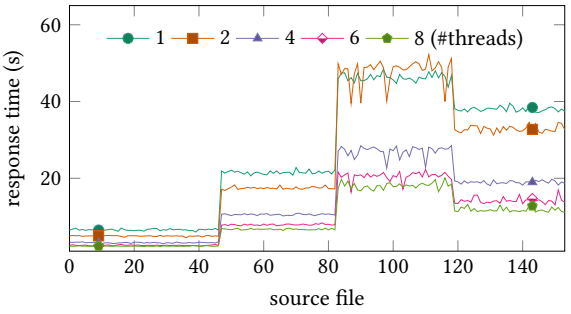
*Results.* Figure 4a shows the response time of the configurations for the benchmark set. Each function graph in the quantile plot refers to a different number of threads used in the analysis. A smaller response time of the analysis corresponds to a smaller state space and relates to a simpler task. The different groups of verification tasks (simple, medium, and difficult) are clearly recognizable by the level of response time, i.e., the plot contains larger steps. Overall, additional threads improve the performance of the analysis.

Figure 4b shows the speedup of our parallel approach over the single-threaded application in the evaluation using box plots. Each entry in the plot shows the median as the horizontal line within the box, together with its two surrounding quartiles between the upper and lower line of the box, as well as the minimum and maximum as whiskers. The speedup becomes larger the more threads we use. The evaluation with 2 threads outperforms the single-threaded execution by about 20% (median). The parallel approach with 8 threads is about three times as fast as with 2 threads.

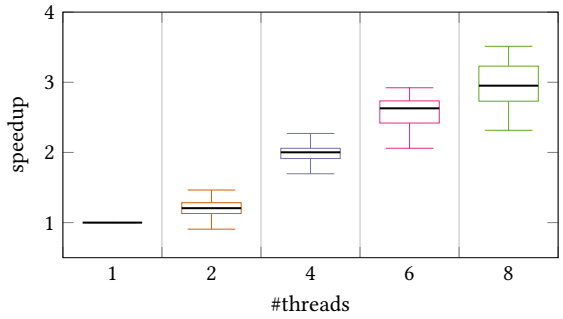
*Discussion.* The results look impressive: Only by parallelizing independent BAM explorations in a way that is not tailored in any specific way towards the framework or to a particular abstract domain, we observe a significant improvement of the response time. Obviously, some parts of the verification process cannot be executed in parallel. This denies a 'perfect' parallelization and is known as Amdahl's law [1]. The sequential parts include the startup process of

[6] A detailed description of quantile plots can be found in the literature [16].

[7] The supplementary artifact [9] and website include additional data about the evaluation of our approach on a benchmark set of tasks containing a property violation.




**(a) Quantile plot for verification tasks without property violation**




**(b) Box plot comparing 1 thread to N threads; without property violation**

**Figure 4: Comparison of response time for different numbers of threads, based on restricted benchmark set**

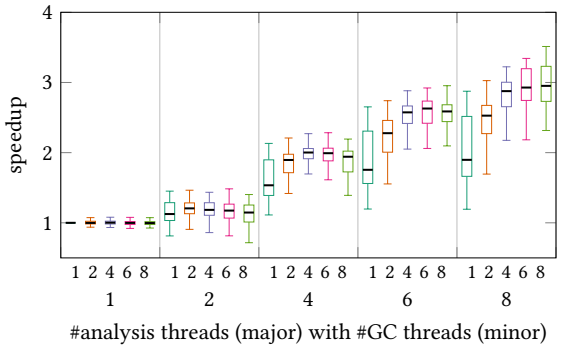
CPAchecker as well as the initial overhead of the analysis to compute blocks for BAM and analyze parts of the most outer block until a nested block is reached, which in turn can be analyzed in a parallel manner. Some parts of the implementation cause an additional synchronization overhead, like multi-threaded statistics for the concurrent access to shared resources like the cache of BAM. The rather modest improvement from 1 thread to 2 threads is most likely due to hyperthreading of the processor, where the two processing units of one physical core share important hardware resources.[8]

We conclude that Claim 2 holds, because the experiments show that for those programs that have potential for speedup by parallelization, we actually observe a significant speedup.

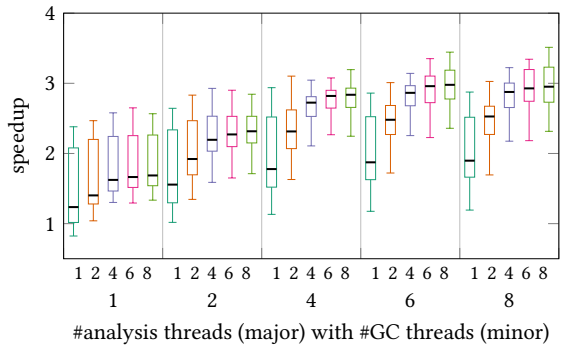
## 4.5 Claim III: Control Influencing Variables

The previous experiments show that several processing units are effectively used by the verification tool, but it is unclear where the benefit comes from. Therefore, we need to investigate which parts of the verifier are parallelized and make sure that our new approach contributed to the benefit. Since our implementation is based on Java, we have also enabled the JVM to use multi-threaded garbage collection (GC), because if we create abstract states in a parallel manner, we should also deallocate them in parallel. The default strategy for GC in OpenJDK 1.8.0 is a combination of *PS MarkSweep* and *PS Scavenge*. The mark-sweep collector applies a full mark-sweep garbage collection algorithm for old-generation objects. The parallel scavenge collector cleans up young-generation objects.

[8] In our experiments we assigned successive processing units to the verification runs; the experiment with 2 threads could be improved by using two processing units of different physical cores.




**(a) Box plot comparing the response time of 1 thread to 8 threads, evaluated on as many processing units as #analysis threads**




**(b) Box plot comparing the response time of 1 thread to 8 threads, evaluated on 8 processing units**

**Figure 5: Comparison of different numbers of analysis threads and different numbers of GC threads**

*Configuration.* We use the 154 tasks from the previous experiment and re-evaluate them. We divide our evaluation into two cases: First, the number of available processing units is equal to the number of analysis threads. Second, the number of available processing units is set to 8, which is the upper limit the available hardware. For both cases, we evaluated all combinations of analysis threads (using 1, 2, 4, 6, 8 threads; major, large numbers in figure) and GC threads (using 1, 2, 4, 6, 8 threads; minor, small numbers in figure).

*Results.* We present the speedup statistics for comparing the response time of a single-threaded analysis with a single-threaded GC on a single processing unit to an execution with a given number of analysis threads with a given number of threads for GC on a given number of processing units. In Fig. 5a the number of available processing units is equal to the number of analysis threads. In Fig. 5b all 8 processing units of the machine are available to the verifier. In both figures we configure the number of analysis threads and GC threads. In each plot, the horizontal axis contains 5 major groups (representing the number of analysis threads) of each 5 minor entries (number of threads for GC). For example, the five first (most left) entries in each figure show the speedup of the approach if using one thread for the analysis and a varying number of threads for GC. Unsurprisingly, the overall result is that using multiple threads for both the analysis and additionally the GC is beneficial. Nearly all tasks are solved faster if multiple processing units are assigned to the verification process.

*Discussion.* In Fig. 5a, the first entry of each group shows the speedup of the analysis when using only one thread for GC. This isolates the the benefit of multi-threading caused by our new analysis approach. Similarly, Fig. 5b shows (within each of the 5 groups) that keeping the number of analysis threads constant and incrementing the number of threads for GC also speeds up the verification process. Therefore both, analysis and GC, benefit from multi-threading. Figure 5b shows that if the analysis is bound to one thread, the benefit from multi-threading is rather limited, while the speedup is improved if we use more threads for the analysis. The most interesting indicators are the median value (middle line inside the box) and the minimal speedup values (lower whisker). The overall variance for the response time and speedup is quite large if there are several processing units available. This might indicate a non-deterministic scheduling of workload across free resources, in contrast to the narrow boxes in Fig. 5a in the left two groups (where the number of processing units is bound to one and two, respectively).

We conclude that Claim 3 holds, because we were able to isolate and control the only other cause for a significant speedup, and the experiments confirmed that our new approach is reponsible for the improved performance of the analysis, while the parallel GC algorithms of the JVM take care of parallelized deallocation.

### 4.6 Threats to Validity

*External Validity:* Our benchmark suite consists of a large set of C source files. We use the largest publicly available benchmark suite in order to optimize the diversity in size and type of programs. This is particularly important for evaluating Claim 1. For Claims 2 and 3, we restricted the benchmark set to verification tasks that have potential to benefit from parallelization. Our evaluation is restricted to the language C, and while it seems clear that the concepts and results can be transferred to other imperative languages, such a claim is not backed up by our experiments. The chosen time limit of 15 min and memory limit of 15 GB for verifying a given task is inspired by the research community on software verification (cf. one of the reports on the International Competition on Software Verification [4]). Of course, the evaluation of our approach depends on the tool in which it is implemented. There is currently no other tool implementing the same approach, and a comparison with a completely different approach for parallel analysis might be misleading. [9] With the assumption that the default configuration is optimized for most use cases, we did not change the configuration of the JVM except the increment of maximal heap memory and the adjustment of the garbage-collection strategy, such that the effect of the number of threads can be measured. The available hardware might also influence the results. For parallel execution, the internal structure of the CPU is a critical element, i.e., low-level caching and the hierarchy of processing units have a large effect on the run time of tasks. We used a modern Intel Xeon E3-1230 v5 that is available on the market for a reasonable price, in order to obtain results that have a higher externally validity than experiments on special high-performance clusters.

[9]The supplementary artifact [9] and website include an additional comparison with some non-BAM analysis approaches, in order to show that using the BAM technology does not negatively effect an analysis' performance (known result [31]).

*Internal Validity:* Besides garbage collection of the JVM, there are other factors that influence the speedup of the parallel approach. Some components of CPAchecker, e.g., counters and measurements for statistics, are not yet fully optimized for parallel execution. Additionally, it depends on the task's structure how many blocks can be analyzed in parallel. Controlling this variable (number of parallelization blocks) is not possible or very difficult, thus, we prefer to increase the internal validity by the large number of experiments on different tasks. Another control variable is the block size. Larger blocks are beneficial for a concurrent analysis, due to the smaller synchronization footprint. For Claims 2 and 3, the benchmark set was already chosen such that it contains only programs where the block size is very large. Thus, we did not further analyze different block sizes. We also need to consider that the explicit-value analysis computes a large number of abstract states, while other abstract domains might lead to more compact representations of the state space, and the fewer abstract states are explored the less might be parallelized. Our time measurement includes the memory allocation for the JVM, parsing time, and internal statistics, which adds processing workload that cannot be parallelized currently. We mitigate this effect by using only those verification tasks that need more than 3 s when using one thread, i.e., we consider verification tasks for which the analysis itself consumes a portion of the run time that is not negligible.

## 5 CONCLUSION

We presented a new approach for multi-threaded software verification that is based on program-block summaries. Our emphasis is on providing a solution that follows the principle of separation of concerns: the problem of making the analysis benefit from multiple processing units is treated completely orthogonal from the problem of designing and implementing an abstract domain and the operators for a program analysis. We formally define the new algorithm in the framework, provide a working implementation, and demonstrate its applicability on a large set of benchmarks. The experiments show that our approach (a) does not add noticeable overhead for verification tasks that do not benefit from parallelization, (b) can considerably speed up the verification process in many cases (given the verification task has a certain minimal size and some independent branches to explore), and (c) contributes largely to the performance improvements, i.e., the speedup is not only due to multi-threading features that the JVM provides.

The presented algorithm is implemented as a shared-memory approach, which allows efficient interaction of all components. As the number of CPU cores per machine and also the amount of memory per host is limited, we plan to extend our algorithm to leverage several processes that might be distributed over several machines in a cluster. An additional benefit would be a simpler usage of abstract domains that rely on libraries that are not thread-safe, because there is no problem with interleaved usage of libraries in separate processes. Additionally we plan to offload the cache of BAM to a disk-based storage, in order to lower the memory usage for very resource intensive tasks. The combination of both, a distributed, multi-process verification algorithm and a disk-based cache, seems to be very promising for the verification of very large programs.

## REFERENCES


[1] G. M. Amdahl. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proc. AFIPS*. ACM, 483–485. https://doi.org/10.1145/1465482.1465560

[2] P. Andrianov, K. Friedberger, M. U. Mandrykin, V. S. Mutilin, and A. Volkov. 2017. CPA-BAM-BnB: Block-Abstraction Memoization and Region-Based Memory Models for Predicate Abstractions (Competition Contribution). In *Proc. TACAS (LNCS 10206)*. Springer, 355–359. https://doi.org/10.1007/978-3-662-54580-5_22

[3] J. Barnat, J. Havlíček, and P. Rockai. 2013. Distributed LTL Model Checking with Hash Compaction. *ENTCS* 296 (2013), 79–93. https://doi.org/10.1016/j.entcs.2013.07.006

[4] D. Beyer. 2017. Software Verification with Validation of Results (Report on SV-COMP 2017). In *Proc. TACAS (LNCS 10206)*. Springer, 331–349. https://doi.org/10.1007/978-3-662-54580-5_20

[5] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. 2016. Correctness Witnesses: Exchanging Verification Results Between Verifiers. In *Proc. FSE*. ACM, 326–337. https://doi.org/10.1145/2950290.2950351

[6] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. 2015. Witness Validation and Stepwise Testification across Software Verifiers. In *Proc. FSE*. ACM, 721–733. https://doi.org/10.1145/2786805.2786867

[7] D. Beyer, M. Dangl, and P. Wendler. 2015. Boosting k-Induction with Continuously-Refined Invariants. In *Proc. CAV (LNCS 9206)*. Springer, 622–640. https://doi.org/10.1007/978-3-319-21690-4_42

[8] D. Beyer, M. Dangl, and P. Wendler. 2018. A Unifying View on SMT-Based Software Verification. *J. Autom. Reasoning* 60, 3 (2018), 299–335. https://doi.org/10.1007/s10817-017-9432-6

[9] D. Beyer and K. Friedberger. 2018. Replication Package for Article "Domain-Independent Multi-threaded Software Model Checking" in Proc. ASE'18. https://doi.org/10.5281/zenodo.1322090

[10] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. 2007. The Software Model Checker Blast. *Int. J. Softw. Tools Technol. Transfer* 9, 5-6 (2007), 505–525. https://doi.org/10.1007/s10009-007-0044-z

[11] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. 2012. Conditional Model Checking: A Technique to Pass Information between Verifiers. In *Proc. FSE*. ACM, Article 57, 11 pages. https://doi.org/10.1145/2393596.2393664

[12] D. Beyer, T. A. Henzinger, and G. Théoduloz. 2007. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Proc. CAV (LNCS 4590)*. Springer, 504–518. https://doi.org/10.1007/978-3-540-73368-3_51

[13] D. Beyer, M. E. Keremoglu, and P. Wendler. 2010. Predicate Abstraction with Adjustable-Block Encoding. In *Proc. FMCAD*. FMCAD, 189–197. https://www.sosy-lab.org/research/pub/2010-FMCAD.Predicate_Abstraction_with_Adjustable-Block_Encoding.pdf

[14] D. Beyer and T. Lemberger. 2016. Symbolic Execution with CEGAR. In *Proc. ISoLA (LNCS 9952)*. Springer, 195–211. https://doi.org/10.1007/978-3-319-47166-2_14

[15] D. Beyer and S. Löwe. 2013. Explicit-State Software Model Checking Based on CEGAR and Interpolation. In *Proc. FASE (LNCS 7793)*. Springer, 146–162. https://www.sosy-lab.org/research/pub/2013-FASE.Explicit-State_Software_Model_Checking_Based_on_CEGAR_and_Interpolation.pdf

[16] D. Beyer, S. Löwe, and P. Wendler. 2017. Reliable Benchmarking: Requirements and Solutions. *Int. J. Softw. Tools Technol. Transfer* (2017). https://doi.org/10.1007/s10009-017-0469-y

[17] D. Beyer and A. Stahlbauer. 2014. BDD-based software verification: Applications to event-condition-action systems. *STTT* 16, 5 (2014), 507–518. https://doi.org/10.1007/s10009-014-0334-1

[18] S. Blom, J. van de Pol, and M. Weber. 2010. LTSmin: Distributed and Symbolic Reachability. In *Proc. CAV (LNCS 6174)*. Springer, 354–359.

[19] M. Brain and F. Schanda. 2012. A Lightweight Technique for Distributed and Incremental Program Verification. In *Proc. VSTTE (LNCS 7152)*. Springer, 114–129. https://doi.org/10.1007/978-3-642-27705-4_10

[20] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50, 5 (2003), 752–794. https://doi.org/10.1145/876638.876643

[21] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. 2011. Practical software model checking via dynamic interface reduction. In *Proc. SOSP*. ACM, 265–278. https://doi.org/10.1145/2043556.2043582

[22] A. Gurfinkel, A. Albarghouthi, S. Chaki, Y. Li, and M. Chechik. 2013. Ufo: Verification with Interpolants and Abstract Interpretation (Competition Contribution). In *Proc. TACAS (LNCS 7795)*. Springer, 637–640. https://doi.org/10.1007/978-3-642-36742-7_52

[23] G. J. Holzmann. 2003. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley.

[24] B. A. Huberman, R. M. Lukose, and T. Hogg. 1997. An Economics Approach to Hard Computational Problems. *Science* 275, 7 (1997), 51–54. http://www.hpl.hp.com/research/idl/papers/EconomicsApproach/EconomicsApproach.pdf

[25] K. Laster and O. Grumberg. 1998. Modular Model Checking of Software. In *Proc. TACAS (LNCS 1384)*. Springer, 20–35. https://doi.org/10.1007/BFb0054162

[26] P. Müller, P. Peringer, and T. Vojnar. 2015. Predator Hunting Party (Competition Contribution). In *Proc. TACAS (LNCS 9035)*. Springer, 443–446.

[27] T. L. Nguyen, P. Schrammel, B. Fischer, S. La Torre, and G. Parlato. 2017. Parallel bug-finding in concurrent programs via reduced interleaving instances. In *Proc. ASE*. IEEE Computer Society, 753–764. https://doi.org/10.1109/ASE.2017.8115686

[28] E. Sherman and M. B. Dwyer. 2018. Structurally Defined Conditional Data-Flow Static Analysis. In *Proc. TACAS, Part II (LNCS 10806)*. Springer, 249–265. https://doi.org/10.1007/978-3-319-89963-3_15

[29] V. Still, P. Rockai, and J. Barnat. 2016. DIVINE: Explicit-State LTL Model Checker (Competition Contribution). In *Proc. TACAS (LNCS 9636)*. Springer, 920–922.

[30] T. van Dijk. 2016. *Sylvan: multi-core decision diagrams*. Ph.D. Dissertation. University of Twente, Enschede, Netherlands. http://purl.utwente.nl/publications/100676

[31] D. Wonisch and H. Wehrheim. 2012. Predicate Analysis with Block-Abstraction Memoization. In *Proc. ICFEM (LNCS 7635)*. Springer, 332–347. https://doi.org/10.1007/978-3-642-34281-3_24


# In-Place vs. Copy-on-Write CEGAR Refinement for Block Summarization with Caching

Dirk Beyer and Karlheinz Friedberger

LMU Munich, Germany

**Abstract.** Block summarization is an efficient technique in software verification to decompose a verification problem into separate tasks and to avoid repeated exploration of reusable parts of a program. In order to benefit from abstraction at the same time, block summarization can be combined with counterexample-guided abstraction refinement (CEGAR). This causes the following problem: whenever CEGAR instructs the model checker to refine the abstraction along a path, several block summaries are affected and need to be updated. There exist two different refinement strategies: a destructive *in-place* approach that modifies the existing block abstractions and a constructive *copy-on-write* approach that does not change existing data. While the *in-place* approach is used in the field for several years, our new approach of *copy-on-write* refinement has the following important advantage: A complete exportable proof of the program is available after the analysis has finished. Due to the benefit from avoiding recomputations of missing information as necessary for *in-place* updates, the new approach causes almost no computational overhead overall. We perform a large experimental evaluation to compare the new approach with the previous one to show that full proofs can be achieved without overhead.

**Keywords:** Software model checking · Block summarization
Copy-on-write · CEGAR · Abstraction refinement · CPAchecker
Program analysis

## 1 Introduction

Software model checking is a powerful technique for proving programs correct as well as for finding errors in programs. Given a program and a specification, a model checker either finds an error path through the program that exposes the specification violation or proves that the specification is satisfied by the program. In this paper, we take a look at the combination of two orthogonal approaches, *block summaries* and *abstraction refinement*.

The technique of constructing summaries of program blocks [18] is effective to reduce the overhead that an exploration without summaries would otherwise


T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11245, pp. 197–215, 2018.
https://doi.org/10.1007/978-3-030-03421-4_14

cause. Block-abstraction memoization (BAM) [27] is based on a standard state-space exploration using a given control-flow automaton (CFA) that represents the program. The CFA is partitioned into blocks, which are analyzed separately by BAM. Block abstractions (e.g., the results of a block's analysis) represent summaries of blocks. Block abstraction is a generalization of function summaries, if the block size is chosen according to function bodies. In general, block abstraction also works for loop bodies and other block definitions. Block abstractions are stored in a cache, such that they can be reused whenever the same block is explored again. The exact behavior of the analysis and the precision of BAM is determined by a wrapped underlying analysis, such as predicate analysis or explicit-value analysis.

Abstraction, i.e., verifying an overapproximating abstract model of the program instead of its concrete state space, is an idea for scaling model checking to large programs orthogonal to summaries. The verification of the abstract model is often less complex and more resource-efficient. Counterexample-guided abstraction refinement (CEGAR) [15] is a property-directed approach for the automatic construction of an abstract model for a given system: it automatically determines a level of abstraction for program verification that is coarse enough to omit unnecessary information from the abstract model and precise enough to refute spurious counterexamples. The basic idea is to iteratively identify relevant facts from infeasible program paths and use them for the further and more precise state-space exploration. Many existing software model-checking algorithms are based on this approach, such as predicate analysis [8], IMPACT [23], and explicit-value analysis [12].

Our combination of block abstraction with CEGAR needs a special refinement strategy such that only the necessary parts of the (cached) state space are touched. However, block abstractions are cached and can be used at different locations during the analysis and even several times on the same error path. The problem is how to correctly refine the block abstractions in the context of BAM based on a the underlying refinement strategy. The original definition of refinement in BAM [27] describes a destructive *in-place* update of block abstractions and explains that *holes* occur in the state space which are caused by modifications on existing block abstractions. Those holes need to be recomputed on demand. However, this is not possible in general, e.g., after the analysis has finished, because the information which block abstraction was computed for which block is no longer accessible. Succeeding analysis steps are not able to recompute the missing information, as not only the block abstractions themselves, but also their dependencies are deleted in the destructive approach. A recomputation would imply to rerun a large part of the complete analysis. Due to the unforeseeable appearance of cache accesses, the recomputation might even produce a completely different counterexample or proof than the previous analysis.

The user usually wants the model checker to terminate with a proof, which in this setting might be an abstract reachability graph (ARG). The ARG is expected to include all initial abstract states and all abstract states that are reachable from the initial abstract states. This guarantee does not hold if the ARG contains holes. Succeeding analysis steps that are executed after the termination of the block-abstraction-based analysis and depend on the full abstract state space (without

holes) have no possibility to recompute the missing parts. For example, correctness witnesses [6,22] can not be reliably produced with block-abstraction-based analyses [1]: the exported correctness witness is either invalid because no graph from root to all reached abstract states could be written, or a missing part in the correctness witness (branch in the graph) is responsible for incorrectly guiding the witness validator.

The main contribution of this paper is a new refinement approach based on a constructive *copy-on-write* strategy. Our work includes a comparative evaluation of the new *copy-on-write* approach with the previous *in-place* refinement, showing that the new approach has only a small computational overhead for run time and memory usage. Because BAM is independent of (and orthogonal to) other analyses in a full program analysis, it can be combined with analyses based on different abstract domains like predicate, value, or interval analysis [11,12], or combinations thereof [1,17]. Our new refinement strategy is fully integrated into BAM in CPAchecker and does not depend on the underlying analysis. Thus, there is no change in the behavior of the sub-analyses.

**Contributions.** We make the following contributions:

- We design a *copy-on-write* approach that solves two open problems: (i) strictly monotonic refinement for summary-based approaches in combination with CEGAR and (ii) abstract reachability graphs without holes that cause problems in later steps of the analysis.
- We implement the approach of *copy-on-write* refinement in the verification framework CPAchecker and make the source code available to others.[1]
- We experimentally evaluate the new approach on a large number of verification tasks to show that the *copy-on-write* approach is about as efficient and effective as the *in-place* approach, although the approach produces *complete* abstract reachability graphs.
- We make all experimental results, including raw data, tables, experiment setup, etc., available on a supplementary web site.[2]

**Related Work.** There are several techniques based on block-based summarization, as this idea dates back to Hoare [20]. The special case of *function summaries* aims at scalability for interprocedural analyses and is integrated in several algorithms and tools.

FunFrog [25,26] uses an SMT solver and Craig interpolation to compute function summaries in the context of bounded model checking. Starting from an initially empty set of function summaries, the tool explores the problem's traces and computes interpolants from path formulas for all missing procedure calls. The interpolants are then directly used as summaries. This strategy is applied in a CEGAR loop until the specified property can be proven or is definitely violated.

[1] https://cpachecker.sosy-lab.org
[2] https://www.sosy-lab.org/research/bam-cow-refinement

FunFrog uses a cache for function summaries, but does never modify existing function summaries.

Bebop [3] and Saturn [28] use binary decision diagrams (BDDs) and SMT to encode the program's semantics. The function summary is build by renaming variables in formulas, such that the direct encoding of a procedure call can be reused several times within the same encoding of the program behavior. Both tools work on a very precise abstraction level and do not refine their summarizations.

BAM is a domain-independent approach for caching and reusing block abstractions. It is independent of functions and can be used with an arbitrary block size. Instead of being limited to a special domain like BDDs, SMT, or intervals, BAM works on an abstract level and can be applied to any abstract domain or even combinations of several domains, including predicate analysis and explicit value analysis [1]. The integration of CEGAR refinement in BAM was already described in the context of predicate analysis [27]. Our new approach of *copy-on-write* refinement for BAM makes the approach really lazy (matching the principles of *lazy abstraction refinement* [19]).

## 2 Background on Block Summarization

The following section provides an overview of some basic concepts and definitions that we use for our approach. We describe the program representation and the most important details of block-abstraction memoization that are used for state-space exploration (cf. other literature on block-abstraction memoization for more detailed descriptions [2,7,27]).

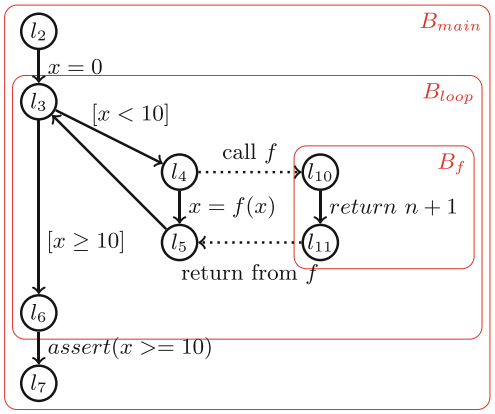
### 2.1 Program and State-Space Representation

A program is represented by a *control-flow automaton* (CFA) $A = (L, l_0, G)$, which consists of a set $L$ of program locations (modeling the program counter), a set $G \subseteq L \times Ops \times L$ (modeling the control flow), and an initial program location $l_0$ (entry point; initial call of the main function). The set $Ops$ contains the operations of the program, i.e., assignment and assume operations, function calls, and function returns. Let $V$ be the set of variables in the program. A *concrete data state* assigns a value to each variable from the set $V$; the set $C$ contains all concrete data states. For every edge $g \in G$, the transition relation is defined by $\overset{g}{\rightarrow} \subseteq C \times \{g\} \times C$. If there exists a sequence of concrete data states $\langle c_0, c_1, ..., c_n \rangle$ with $\forall i \in [1, n] : \exists g : c_{i-1} \overset{g}{\rightarrow} c_i \wedge (l_{i-1}, g, l_i) \in G$, then state $c_n$ is called *reachable* from $c_0$ for $l_0$, i.e., there exists a syntactic walk through the CFA.

We perform a reachability analysis that unrolls the program lazily [19] into an *abstract reachability graph* (ARG) [8]. An ARG $S = (N, E)$ is a directed acyclic graph, consisting of a set $N$ of ARG nodes (representing the abstract program states, e.g., including program location and variable assignments) and a set $E \subseteq N \times N$ of edges modeling the transfer that leads from one abstract state to the next one. We define a *subgraph* $S_s = (s, N_s, E_s)$ as a connected component of an ARG $S = (N, E)$, starting at a given abstract state $s \in N_s$ (denoted as root), such that $N_s \subseteq N$, $E_s \subseteq E$, and $\forall s' \in N_s : (s', s'') \in E \Rightarrow (s'' \in N_s \wedge (s', s'') \in E_s)$.

```
1  void main() {
2      int x = 0;
3      while (x<10) {
4          x = f(x);
5      }
6      assert(x>=10);
7  }
8
9  int f(int n) {
10     return n+1;
11 }
```

**(a)** Source code




**(b)** CFA with blocks

**Fig. 1.** Example program and its control-flow automaton with 3 blocks

### 2.2 Block Summarization

Block-abstraction memoization (BAM) [27] is a generalization of several block-based summarization approaches [18,24,25]. BAM divides an input program into smaller parts, named *blocks*, to analyze them separately by summary construction. It uses an arbitrary block size and is not limited to function boundaries. In addition, BAM uses a cache to reuse block abstractions. The blocks allow us to abstract from the surrounding context, reducing computational overhead, and improving the performance of an analysis. The analysis of each block corresponds to an abstract initial state at the block-entry location and a set of abstract exit states at the block-exit locations (both described later). Block abstractions (e.g., the combination of initial states and exit states of a block's analysis) are stored in a cache, such that they can be reused whenever the same block is visited again.

**Blocks.** The basic components of BAM are *blocks*, which are formally defined as parts of a program: A block $B = (L', G')$ of a CFA $A = (L, l_0, G)$ consists of a set $L' \subseteq L$ of connected program locations and a set $G' = \{(l_1, op, l_2) \in G \mid l_1, l_2 \in L'\}$ of control-flow edges. Two different blocks $B_1 = (L'_1, G'_1)$ and $B_2 = (L'_2, G'_2)$ are either disjoint ($L'_1 \cap L'_2 = \emptyset$) or one block is completely nested in the other block ($L'_1 \subset L'_2$). Each block $B = (L', G')$ has *entry* and *exit locations*, which are defined as $In(B) = \{l \in L' \mid (\exists (l', op, l) \in G \wedge l' \notin L') \vee \nexists (l', op, l) \in G\}$ and $Out(B) = \{l \in L' \mid (\exists (l, op, l') \in G \wedge l' \notin L') \vee \nexists (l, op, l') \in G\}$, respectively. In general, the block size can be freely chosen in BAM. In most cases, function and loop bodies are taken as blocks, because they represent logical structures of the program and seem to be a good choice for block abstraction.

Figure 1 shows the CFA (b) for an example program (a). The CFA is structured into three nested blocks $B_{main}$, $B_{loop}$, and $B_f$, such that their sizes align with the

function and loop bodies. In the example, each block has only one entry and one exit location, e.g., $In(B_{main}) = \{l_2\}$, $Out(B_{main}) = \{l_7\}$, $In(B_{loop}) = \{l_3\}$, $Out(B_{loop}) = \{l_6\}$, $In(B_f) = \{l_{10}\}$, and $Out(B_f) = \{l_{11}\}$.

**State-Space Exploration with BAM.** BAM is an algorithm for program analysis that makes use of a wrapped program analysis $\mathbb{W}$, which tracks data facts and does the *actual* (block-local) program-analysis work, i.e., computes abstractions, formulas for paths, or checking whether the property holds. Our framework is based on the concept of configurable program analysis (CPA) [9] and uses, for example, predicate analysis (based on SMT solving and predicates), explicit-value analysis (tracks assignments of variables), or combinations thereof, with usage of common basic components such as location analysis (tracks the program counter) or call-stack analysis (tracks the current call stack). Each CPA provides the analysis operators, like the transfer relation $\rightsquigarrow$ to compute abstract successor states for a specific abstract domain. BAM is specified as a CPA and does not know about the internals of the wrapped analysis $\mathbb{W}$, which is also a CPA. The approach of BAM just operates on abstract states of a (possibly combined) abstract domain to generate block abstractions.

The state-space exploration with BAM is defined recursively for blocks. The successor computation for abstract states chooses between two possible steps, depending on the currently analyzed program location: At an entry location of a block $B$, the successor computation $\overset{B}{\rightsquigarrow}_{\mathbb{B}}$ of the containing block executes a separate *nested sub-analysis* of the block $B$ (starting with the initial abstract state for the block-entry location). This step produces a separate ARG that is later integrated as block abstraction into the surrounding analysis context. The block abstraction can either be computed or taken from a cache, if the block has been analyzed before. For block-exit locations of blocks, there is no succeeding abstract state (in the nested sub-analysis). For other program locations, the successor computation $\rightsquigarrow_{\mathbb{W}}$ is applied, which acts according to the abstract domain of the wrapped analysis $\mathbb{W}$ (e.g., tracks variables or computes abstractions for abstract states). Abstract states where a specification violation occurs are handled as if those abstract states are at block-exit locations of the current block, i.e., the nested sub-analysis terminates and returns the violating abstract states directly for the block abstraction.

Note that an ARG can contain edges representing block abstractions. The block of the block abstraction is inlined whenever a concrete program path without block abstractions is needed. This overhead is the necessary price for having a block-modular analysis. When CEGAR modifies the ARGs during the refinement, a problem occurs, which we will describe later.

### 2.3 CEGAR

Counterexample-guided abstraction refinement (CEGAR) is an approach to automatically adjust the granularity of an analysis by learning from infeasible error paths the relevant analysis facts that are needed to verify a program. We use

CEGAR as a wrapper algorithm around the state-space exploration algorithm, which is implemented as CPA algorithm [10]. The granularity of the analysis is defined as a *precision* that is refined in each iteration of the CEGAR algorithm. Each abstract state in an ARG has a precision. The precision of an abstract state can be changed during the refinement step. A too coarse precision would lead to an imprecise analysis that reports false alarms, a too fine precision would lead to an expensive state-space exploration; CEGAR tries to find the "right" level of abstraction in between.

CEGAR consists of two steps, an exploration step and a refinement step, which are executed alternatingly until a feasible error path is found (and a bug is reported) or all error paths are proven to be infeasible (and a proof can be reported): The exploration step computes new successor abstract states and builds the abstract state space in form of an ARG $G = (N, E)$, using the level of abstraction determined by CEGAR. When finding a possible specification violation, a feasibility check is applied, which examines the error path to the violation. A feasible error path is reported and the analysis terminates. An infeasible error path is used for a refinement step to gain more relevant facts from the program, e.g., by applying interpolation, and refine the precision. If the exploration step does not find any property violation and all abstract states are explored, the algorithm terminates and the program is proven correct.

The refinement step determines a *cut point* $s_{cut} \in N$ and a new precision for this position, such that the new level of abstraction is sufficient to exclude the infeasible error path from further exploration of the state space. The level of abstraction depends on the abstract domain of the analysis and might consist of, e.g., predicates (for predicate analysis) or a set of variables to be tracked (for value analysis). The outdated (too imprecise) subgraph $S_{s_{cut}} = (s_{cut}, N', E')$ of the already explored state space is removed from the ARG $G$ and the subgraph's root state $s_{cut}$ alone is re-added to $G$, such that the next exploration step of CEGAR recomputes this part of the state space with a higher precision. There are several approaches to determine the cut point $s_{cut}$ along the error path [13]:

- **cut point at root:** full eager refinement is applied, where the whole explored state space is withdrawn and re-exploration starts from the initial root state of the ARG (e.g. [4,14,16]),
- **cut point as deep as possible:** only a (minimal) part of the explored state space is removed (lazy refinement [19]), such that a large part of the explored state space remains intact and can be reused in the further analysis (e.g. [8,10]), or
- **cut point in between:** trade-off between reuse and reexploration is somewhere in between the above two choices [13].

The second approach performs best in most cases and is currently used in the field. As shown in Alg. 1, the refinement procedure first determines an abstract state $s_{cut}$ where the infeasible subgraph is to be cut off and new facts are applied to the precision of the analysis. Lazy refinement is based on the idea that some parts of a program are analyzed with a coarse abstraction level and only some

**Algorithm 1.** Default refinement procedure of CEGAR

**Input:** an infeasible error path $\sigma$, an ARG $G$ of the analysis
$s_{cut}, newFacts := \mathsf{refine}_{\mathbb{W}}(\sigma)$
$\mathsf{refinePrecision}(G, s_{cut}, newFacts)$
$\mathsf{removeSubgraph}(G, s_{cut})$

other parts of a program use a more fine-grained precision. Cutting off only a part of the ARG in each refinement fulfills this requirement.

### 2.4 Requirement for Refinement Approaches: New Precision Strictly More Precise

We use a (partial) order on precisions, such that a precision is considered as *more precise* compared to another precision, if it causes the analysis to track more information. For example, if an analysis uses a precision to track a set of variables or predicates (as predicate analysis and value analysis do), this relation is implicitly given by the subset relation. If a precision $p$ is a superset of another precision $p'$, then $p$ is *more precise* than $p'$. Let an infeasible error path be a sequence of abstract states $\langle s_0, s_1, ...s_n\rangle$ with their precisions $\langle p_0, p_1, ...p_n\rangle$, such that $s_0$ is the root of the program and $s_n$ violates the specification. The sequence $\langle p_0, p_1, ...p_n\rangle$ of precisions is *more precise* then a sequence $\langle p'_0, p'_1, ...p'_n\rangle$ of precisions if either $p_0$ is more precise than $p'_0$, or $p_0 = p'_0$ and the remaining sequence $\langle p_1, ...p_n\rangle$ of precisions is *more precise* than the sequence $\langle p'_1, ...p'_n\rangle$. We require a *refined* precision to be strictly more precise than its original, in order to guarantee *progress* in CEGAR (monotonic refinement).

Since CEGAR is a fixed-point algorithm that starts with a coarse precision and refines it until it is sufficiently precise to prove or refute the program, the termination criterion for the CEGAR loop depends on a refinement approach that monotonically increases the precision. To ensure progress of the analysis, the refinement requirement needs to hold for each single refinement step in a program analysis with CEGAR.

Removing a subgraph $S_{s_{cut}} = (s_{cut}, N', E')$ from an ARG and applying a refined precision at its cut point $s_{cut}$ fulfills the property, because the precision itself is more precise for the cut point, the predecessors are not touched, and the successors are deleted (and implicitly inherit the refined precision). Even if the removed subgraph $S_{s_{cut}}$ contained a more precise precision for some abstract state, the refinement requirement holds: Because the refined precision is represented as mapping from locations to precisions, and assigned as precision of the root abstract state $s_{cut}$ of the subgraph, an ancestor of any removed state will be seeded with the new, refined precision. During re-exploration of the deleted subgraph, the analysis will re-explore prefixes of previously encountered error paths in this part of the state space and perform refinements of other error paths with cut points that also satisfy the refinement requirement. Strengthening the precision by additional information (like invariants from an external tool) before applying the update during the refinement also fulfills the property.

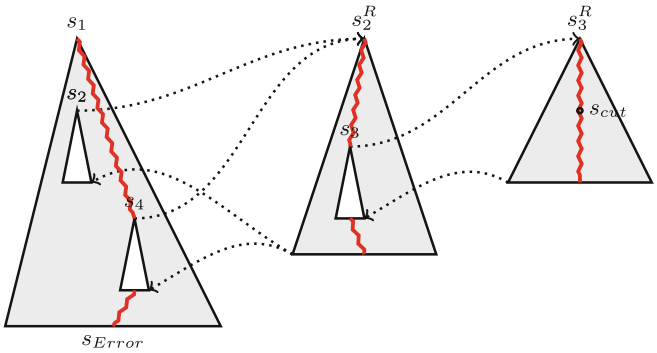



**Fig. 2.** State-space exploration with BAM and cut points for refinement

The refinement requirement is not fulfilled by the *in-place* approach, because the *in-place* refinement potentially deletes block abstractions for already analyzed parts of the state space and causes additional overhead if recomputation is needed for those missing block abstractions. The *copy-on-write* approach does not suffer from this problem.

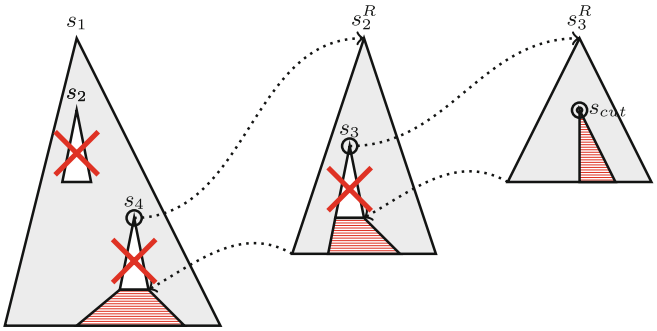
After defining all necessary parts, we will now go on with a motivating example, before giving the detailed description of the refinement approaches in BAM. Our goal is to replace an implementation that *in-place* modifies the ARG by a new *copy-on-write*-based approach for modifying the ARG. This allows us to efficiently keep the original as well as the copy for further processing. In the later evaluation we show that keeping the original data improves our analysis in several cases, and in particular, leaves the ARG complete (without holes).

## 3 Motivating Example

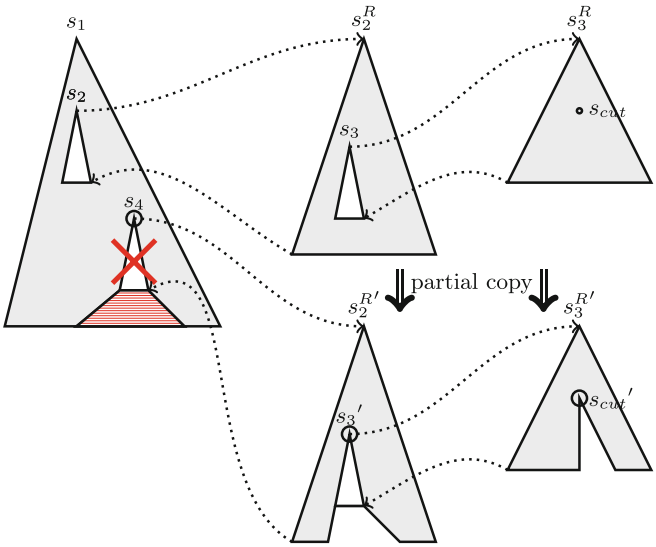
The following example illustrates the differences of the two strategies that could be used as refinement step in a CEGAR approach. In BAM, the analysis explores the state space and computes block abstractions for blocks. An example for such a state-space exploration is given in Fig. 2 (gray triangles represent ARGs, rooted at the top corner; white triangles represent block abstractions). We use block abstractions for nested blocks at the entry abstract states $s_2$, $s_3$, and $s_4$ with the corresponding initial abstract states $s_2^R$, $s_3^R$, and $s_4^R$ in the nested analysis for the blocks at those program locations. In the example, let $s_4^R$ be equal to $s_2^R$, such that we can reuse the existing block abstraction here. Block abstractions are shown as white triangles and are connected with their ARG via dotted lines. When finding the property violation $s_{Error}$, the analysis stops and performs a refinement for the found counterexample. The lazy refinement approach determines a possible cut-state, i.e., an abstract state $s_{cut}$ along the error path, from where the found property violation is no longer reachable if a refined precision is applied.

At this point, the two refinement strategies differ:

Figure 3a shows the *in-place* refinement, removing parts of the explored state space, i.e., everything after the cut point $s_{cut}$ and after the block abstractions (for




**(a)** Strategy 1: In-place refinement step with removed block abstractions and subgraphs




**(b)** Strategy 2: Copy-on-write refinement step with copied ARGs and changes only in the most outer ARG

**Fig. 3.** In-place and copy-on-write refinement approach for BAM

abstract states $s_3$ and $s_4$). At the abstract state $s_2$ the *in-place* approach implicitly invalidates the ARG for the block abstraction and causes a *hole* in the surrounding ARG. Here, the applied block abstraction itself remains valid, because those abstract states were computed before the refinement.

Figure 3b shows the *copy-on-write* approach, updating the abstract states. All inner (nested) ARGs are updated *copy-on-write*. The red horizontal lines represent the removed abstract successor states after the block abstraction for abstract state $s_4$. The ARGs rooted at $s_2^R$ and $s_3^R$ are copied into new ARGs with roots

at $s_2^{R'}$ and $s_3^{R'}$, leaving out the parts that are invalid after updating the precision. The references to or from block abstractions are also updated.

The difference in the refinement strategies is visible in Figs. 3a and 3b. While the first approach deletes and recomputes parts of the ARGs, the second approach works on fresh copies of the ARGs and uses them along with the old ARGs. In the following we explain both refinement strategies in more detail and discuss benefits of the second approach.

## 4 *In-Place* Refinement for BAM

The existing approach for refinement in BAM (as described earlier [27]) is sound, simple, and efficient, but has problems when abstract states need to be accessible afterwards. Briefly worded, the existing approach modifies cached block abstractions *in-place* and deletes important information that is not available after the refinement and needs to be recomputed for further steps of the analysis.

### 4.1 *In-Place* Refinement Algorithm for BAM

Algorithm 2 gives an overview of the *in-place* refinement of CEGAR for BAM, without going into detail for the further operation of BAM itself (cache management). The *in-place* refinement tries to mimic lazy abstraction refinement and CEGAR, i.e., it touches only a small number of abstract states and aims to update only those states where a precision update will avoid the re-exploration of the currently found infeasible error path. In contrast to Alg. 1, there is no single ARG $G$ to work on, but with BAM there are several ARGs and the refinement must be applied to several of them. Algorithm 2 applies the following steps of the refinement:

After the refinement procedure of the underlying analysis has computed new facts for the analysis and determined an abstract state $s_{cut}$ along the error path, the refinement approach determines the subgraph $S = (N, E)$ where the cut point $s_{cut}$ is located. BAM might have used the block abstraction for $S$ several times along the error path, and thus, we need to find out which outer subgraphs we need to remove (see Fig. 3a). Thus, we start from the correct block abstraction, and apply the removal operations for *in-place* refinement: The cut point $s_{cut}$

**Algorithm 2.** In-place refinement procedure of CEGAR with BAM

**Input:** an infeasible error path $\sigma$
$s_{cut}, newFacts := \mathsf{refine}_{\mathbb{W}}(\sigma)$
$S := \mathsf{getARG}(s_{cut}, \sigma)$
$\mathsf{refinePrecision}(S, s_{cut}, newFacts)$
$\mathsf{removeSubgraph}(S, s_{cut})$
**while** $S$ is nested in another ARG $S^*$ along $\sigma$ **do**
  $s^* := \mathsf{getInitState}(S, S^*, \sigma)$
  $\mathsf{removeSubgraph}(S^*, s^*)$
  $S := S^*$

and its subgraph $S_s(s_{cut}, N_s, E_s)$ get removed from $S$ (with $s_{cut} \in N_s \subseteq N$ and $E_s \subseteq E$).

If the ARG $S$ represents a block abstraction for a block $B$, i.e., $S$ is nested within another ARG $S^* = (N^*, E^*)$, with the ARG $S$ rooted at the initial abstract state $s^* \in N^*$, the subgraph $S_{s^*}$ starting at the abstract state $s^*$ of the nested-block abstraction is removed from the surrounding ARG $S^*$. This strategy is applied transitively up to the most outer ARG. The most outer block is not used as a block abstraction (it represents the whole program) and thus never referred to elsewhere in the state space.

The succeeding exploration step of CEGAR will re-explore the removed parts, use or recompute block abstractions and reach the abstract state with the refined precision, from where the state space is analyzed without exploring the previously encountered infeasible error path. Every ARG modification happens *in-place* and directly modifies the existing block abstractions. This approach does not consider whether a block abstraction was already used in another part of the state space, e.g., as part of another another ARG.

### 4.2 Problem of Cached Block Abstractions with *In-Place* Updates

The *in-place* refinement approach suffers from the *in-place* update of block abstractions in the following way: Whenever an missing abstract state belonging to a *hole* (missing block abstraction) in the state space is needed to be accessed, e.g., as part of a new error path, the block-entry state of the *hole*'s block abstraction is determined (depending on the context) and a possible valid block abstraction is recomputed. The previously updated ARG can not be used to fill the *hole*, because its precision might have been refined and updated *in-place*, such that it is more precise than before and leads to different block-exit abstract states. In order to not loose this refined precision for further exploration, all abstract states following the recomputed block abstraction need to be replaced by their recomputed counterparts (which also happens *in-place*).

In Fig. 3a this case happens when a property violation is found with an error path going through the removed block abstraction of $s_2$. Then the subgraph of $s_2$ needs to be removed and recomputed with a new block abstraction.

After BAM terminates (with or without finding a property violation) we often generate statistics or collect some data from the reached abstract state space. However, with *holes* there also comes the problem of missing data. This is only a minor problem, however might also irritate and mislead the user. Numeral statistics like the *number of abstract states* or the *number of predicates* are potentially misleading. Missing parts in non-numerical output, such as proofs and correctness witnesses, cause problems for later processing of the verification results. For example, we have identified several tasks in SV-COMP'17, for which CPAchecker (competition contribution BAM-BnB [1]) computed the correct result during the analysis, but did not write a correctness witness for the validation, or a witness was written, but the graph of the witness was missing some parts, such that the witness validator was not correctly guided to some branches and could not successfully validate the result.

The new *copy-on-write* approach does not suffer from these problems, because the necessary data are kept until it is no longer needed, and we obtain correct statistics and valid (and complete) correctness witnesses.

## 5 *Copy-on-Write* Refinement for BAM

This section describes our new approach for *copy-on-write* refinement in BAM and considers the computational difference to *in-place* refinement.

### 5.1 *Copy-on-Write* Algorithm for BAM

We define a *copy* of an ARG $S = (N, E)$ as a second graph $S' = (N', E')$, where each ARG abstract state from $N$ and transition from $E$ is copied into $N'$ and $E'$. Technically, a copy is just a new instance of the same ARG. Instead of changing an existing ARG $S$, whenever we would need to remove a subgraph $S_s$ from it, the *copy-on-write* algorithm (Alg. 3) copies the ARG $S$ into a new ARG $S'$, omitting the corresponding subgraph. Then, we update the precision only for abstract states in the new instance $S'$. The new ARG $S'$ is then registered in the cache as new block abstraction for one position where previously $S$ was used, such that further explorations use the new instance $S'$. The old ARG $S$ remains untouched, is still valid, and can be accessed when revisiting existing block abstractions.

When copying an ARG $S$ that contains an embedded ARG $S^{nested}$ from a nested sub-analysis, Alg. 3 only references the existing instance $S^{nested}$ in the new ARG $S'$ and does not copy it, except $S^{nested}$ itself has to be modified. In this case, a copy $S^{nested\prime}$ is inserted instead of the original $S^{nested}$.

**Computational Overhead for *Copy-on-Write* Refinement.** The run time of the *copy-on-write* refinement is similar to the *in-place* approach, because every affected ARG is exactly traversed once in each of the approaches. Thus, the run time of both refinement strategies is linear in the number of reached states. The conceptional difference comes with the operation performed on the affected abstract states: Instead of removing a subgraph of abstract states from an ARG, we create a flat copy of all other abstract states, i.e., those abstract states that are not part of the subgraph.

**Algorithm 3.** Copy-on-write refinement procedure of CEGAR with BAM

**Input:** an infeasible error path $\sigma$

$s_{cut}, newFacts :=$ refine$_{\mathbb{W}}(\sigma)$
$S :=$ getARG$(s_{cut}, \sigma)$
$S', s'_{cut} :=$ copyWithoutSubgraph$(S, s_{cut})$
registerARG$(S')$
refinePrecision$(S', s'_{cut}, newFacts)$
**while** $S'$ is nested in another ARG $S^*$ along $\sigma$ **do**
  $S^* :=$ getInitState$(S', S*, \sigma)$
  $S^{*\prime}, s^{*\prime} :=$ copyWithoutSubgraph$(S^{*\prime}, s^*)$
  registerARG$(S^{*\prime})$
  $S' := S^{*\prime}$

To reduce the run time of the copy operation and the memory footprint for the copied abstract states, the *flat copy* keeps all internal data of abstract states untouched (e.g., information about program location, call stack, data state, etc.) and just references them from the new abstract states. This perfectly matches the *copy-on-write* idea and also the internal data structure of our framework, where abstract states consist of separate components for separate domains. Only those components where data needs to be changed are effectively constructed again, the rest is just referenced. This approach has two benefits:

- There is no need to implement and execute methods for copying internal data of abstract states (predicates, variable assignments, program counter, call-stack information, ...). Thus our new approach can easily be applied to all existing analyses.
- The *copy-on-write* approach has only a small memory overhead, because only new ARG states are constructed, the internal data of abstract states are shared and do not require additional memory.

## 6 Evaluation

Next we give evidence that the improvements in our refinement approach (no holes in the ARG) do not lead to significant performance drawbacks.

**Benchmark Set.** We evaluate our new *copy-on-write* refinement approach on a large subset of the SV-benchmark suite[3] containing over 5 500 verification tasks and compare it with the existing *in-place* approach.

**Setup.** We run all our experiments on computers with Intel Xeon E3-1230 v5 CPUs with 3.40 GHz, and limit the CPU time to 15 min and the memory to 15 GB. We use our implementation in CPAchecker[4] in revision `r29066`. The time needed for parsing the input program and exporting data is rather small compared to the analysis time, thus we measure the complete CPU time for the verification run of CPAchecker (i.e., including parsing, analysis, and witness export).

**Analysis Configuration.** BAM can be combined with several analyses and for our experiments, we choose two combinations that are used in practice: BAM with predicate analysis (PA) and BAM with value analysis (VA) [2]. We configure BAM to use function and loop bodies as blocks, and predicate analysis computes abstractions, just as in the original work [27]. The expressive power of the program analysis depends only on the expressiveness of the predicate analysis or value analysis, and is not influenced by BAM. Except for the refinement approach itself, we do not change any configuration for each of the analyses. Thus, each of analyses should give the same verification answer in both cases.

[3] https://github.com/sosy-lab/sv-benchmarks
[4] https://cpachecker.sosy-lab.org

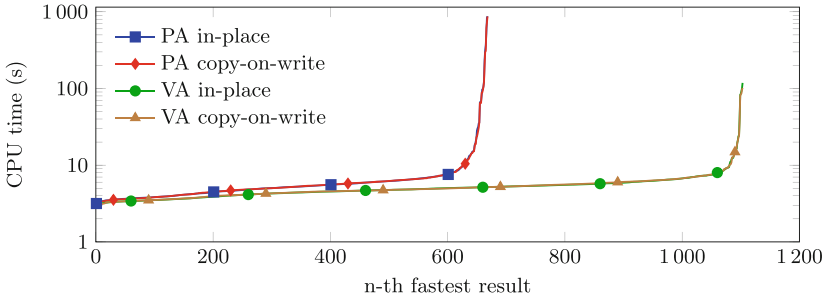
**Results and Discussion.** The experiments show nearly no difference in CPU time and also no significant difference in memory consumption between the two refinement approaches for each analysis. The reason for this result in terms of run time is that *copy-on-write* is extremely efficient and the light overhead is compensated by savings for recomputing missing parts of the state space. The reason for the same memory footprint is that the memory overhead for the additional ARGs is very small compared to the shared data (e.g., formulas for tracking variables). Note that memory usage is not fully predictable in general, as the Java garbage collection is applied non-deterministically.

The quantile plots in Fig. 4 show how many tasks are solved correctly with each of the approaches and each of the analyses. Figure 4a presents the results for all correctly solved verification tasks with low number of refinements ($\leq$1): no difference in the results is visible for the two approaches per underlying analysis. For a low number of refinements, the equality of the results for different refinement approaches was expected, because the effect of missing block abstractions depends on a sufficiently large number of refinements. With only zero or one refinement the new approach behaves exactly as the *in-place* approach. With a growing number of refinements, the analysis could in principle perform differently. Figure 4b shows the CPU time for all correctly solved verification tasks where more than one refinement was needed. Both refinement approaches perform very similar, e.g., keeping block abstractions using *copy-on-write* is as good as recomputing missing block abstractions for both underlying analyses. (The similar performance of predicate analysis and value analysis is a coincidence, because both analyses use completely different techniques to track variables, assignments, and relations.)

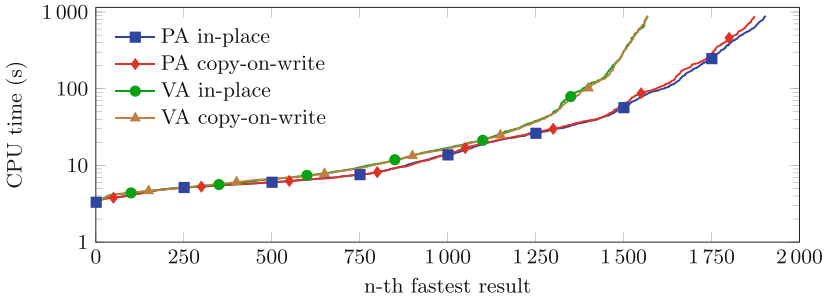
Table 1 shows statistics about all verification results, for both approaches. There are some cases (for both predicate analysis and value analysis), where the analysis with one refinement approach delivers a result while the other does not. Sometimes eager application of a refined precision is beneficial, sometimes the overhead for recomputation of a missing block abstraction is too expensive. While the difference for value analysis is negligible, predicate analysis performs better with the *in-place* refinement, but needs more refinements than with *copy-on-write*. It seems that predicate analysis reacts much more fragile to changes in the refinement strategy and application of refined precisions than value analysis.

Figures 5a and 5b compare the number of needed refinements for each solved task using scatter plots. The number of refinements includes also cases where a missing block abstraction has to be recomputed (recomputation is lazy and only applied if an error path with a *hole* was found; thus it counts as refinement, too). For predicate analysis with the *copy-on-write* approach, the majority of results is computed with a smaller number of refinements than with the *in-place* refinement: on average the new approach needs only a third of the refinements. For value analysis there is no clear difference in the number of refinements and the average number of refinements is also similar.

**Threats to Validity.** Our evaluation uses a large publicly available benchmark suite of C verification tasks in order to optimize the diversity in size and type of




**(a)** CPU time of refinement approaches of BAM, $\leq 1$ refinements only (plots are identical, because the changed approach does not affect the analysis)

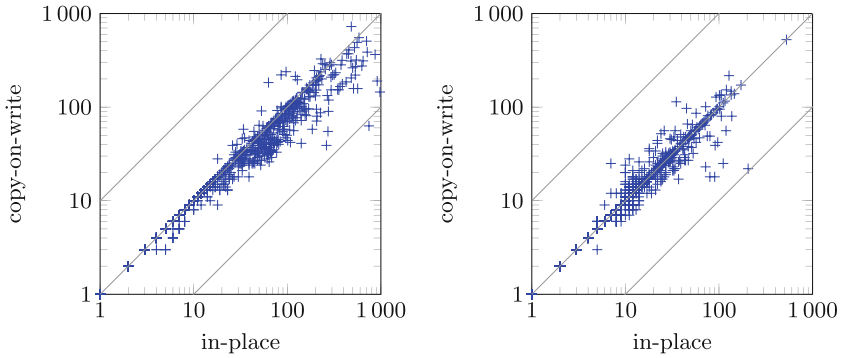



**(b)** CPU time of refinement approaches of BAM, $\geq 2$ refinements only (plots only differ for predicate analysis with a larger run time, e.g., over 200 s, because the changed approach only affects the analysis if several blocks are analyzed repeatedly and the cache is accessed)

**Fig. 4.** Quantile plots for CPU time of refinement approaches

**Table 1.** Statistics of refinement approaches of BAM

| | Predicate Analysis | | Value Analysis | |
|---|---|---|---|---|
| | In-place | Copy-on-write | In-place | Copy-on-write |
| Found proofs | 2149 | 2121 | 2352 | 2352 |
| Found bugs | 425 | 422 | 322 | 322 |
| Incorrectly found proofs | 2 | 2 | 0 | 0 |
| Incorrectly found bugs | 0 | 0 | 2 | 2 |
| Solved by only one approach | 40 | 9 | 4 | 4 |
| Avg. no. of refinements | 53.7 | 19.4 | 8.21 | 8.35 |




**(a)** #Refinements for predicate analysis **(b)** #Refinements for value analysis

**Fig. 5.** Comparison of *in-place* and *copy-on-write* refinement approach for predicate analysis and value analysis

programs. While it seems clear that the concepts and results can be transferred to other verification tasks, such a claim is not backed up by our experiments. Besides the internal structure of a verification task, there are other factors that influence the behavior of an analysis. Thus, the external validity of the experiments regarding the application of refinements and precision updates is increased by the large number of experiments on different tasks. The chosen time limit of 15 min and memory limit of 15 GB for verifying a given task is inspired by the research community on software verification (cf. one of the reports on the International Competition on Software Verification [5]). Of course, the evaluation of our approach depends on the tool where it is implemented. To our knowledge, there is no other tool directly implementing the approach of BAM.

## 7 Conclusion

We developed a new approach for CEGAR-based refinement of block summaries that is based on *copy-on-write*. The new approach makes it possible to construct an abstract reachability graph without holes, such that at the end of the program analysis, a complete proof is available to the user. The proof can be dumped for inspection, or a correctness witness can be extracted from the proof. We designed and implemented the *copy-on-write* refinement and provide a ready-to-use implementation in the framework CPAchecker. Re-using existing underlying analyses is possible without any further development overhead. The experimental comparison showed that there is almost no performance overhead for copy-on-write. Furthermore, the experimental comparison of the existing *in-place* with the new *copy-on-write* refinement strategy revealed interesting insights into some aspects of block summarization. In the future, we plan to design a parallel version of BAM to utilize a network of computers for our domain-independent analysis technique (cf. Swarm [21]): The new immutable block abstractions might also be beneficial in the context of resource-intensive communication between nodes of a computer network.

## References


1. Andrianov, P., Friedberger, K., Mandrykin, M.U., Mutilin, V.S., Volkov, A.: CPABAM-BnB: Block-abstraction memoization and region-based memory models for predicate abstractions. In: Proc. TACAS. LNCS, vol. 10206, pp. 355–359. Springer (2017)
2. Andrianov, P., Mutilin, V.S., Mandrykin, M.U., Vasilyev, A.: CPA-BAM-Slicing: Block-abstraction memoization and slicing with region-based dependency analysis (competition contribution). In: Proc. TACAS. LNCS, vol. 10806, pp. 427–431. Springer (2018)
3. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for boolean programs. In: Proc. SPIN. LNCS, vol. 1885, pp. 113–130. Springer (2000)
4. Ball, T., Rajamani, S.K.: The Slam project: Debugging system software via static analysis. In: Proc. POPL, pp. 1–3. ACM (2002)
5. Beyer, D.: Software verification with validation of results (Report on SV-COMP 2017). In: Proc. TACAS. LNCS, vol. 10206, pp. 331–349. Springer (2017)
6. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE, pp. 326–337. ACM (2016)
7. Beyer, D., Friedberger, K.: Domain-independent multi-threaded software model checking. In: Proc. ASE, pp. 634–644. ACM (2018)
8. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transfer **9**(5–6), 505–525 (2007)
9. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proc. ASE, pp. 29–38. IEEE (2008)
10. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. LNCS, vol. 6806, pp. 184–190. Springer (2011)
11. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD, pp. 189–197. FMCAD (2010)
12. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE. LNCS, vol. 7793, pp. 146–162. Springer (2013)
13. Beyer, D., Löwe, S., Wendler, P.: Refinement selection. In: Proc. SPIN. LNCS, vol. 9232, pp. 20–38. Springer (2015)
14. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Trans. Softw. Eng. **30**(6), 388–402 (2004)
15. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003)
16. Clarke, E.M., Kröning, D., Sharygina, N., Yorav, K.: SatAbs: SAT-based predicate abstraction for ANSI-C. In: Proc. TACAS. LNCS, vol. 3440, pp. 570–574. Springer (2005)
17. Friedberger, K.: CPA-BAM: Block-abstraction memoization with value analysis and predicate analysis. In: Proc. TACAS. LNCS, vol. 9636, pp. 912–915. Springer (2016)
18. Henzinger, T.A., Jhala, R., Majumdar, R.: Race checking by context inference. In: Proc. PLDI, pp. 1–13. ACM (2004)
19. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL, pp. 58–70. ACM (2002)
20. Hoare, C.A.R.: Procedures and parameters: An axiomatic approach. In: Symposium on Semantics of Algorithmic Languages, pp. 102–116. Springer (1971)
21. Holzmann, G.J., Joshi, R., Groce, A.: Tackling large verification problems with the Swarm tool. In: Proc. SPIN 2008. LNCS, vol. 5156, pp. 134–143. Springer (2008)

22. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Comput. Sci. Rev. **5**(2), 119–161 (2011)
23. McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. CAV. LNCS, vol. 4144, pp. 123–136. Springer (2006)
24. Reps, T.W.: Program analysis via graph reachability. In: Proc. ILPS, pp. 5–19. MIT (1997)
25. Sery, O., Fedyukovich, G., Sharygina, N.: Funfrog: Bounded model checking with interpolation-based function summarization. In: Proc. ATVA. LNCS, vol. 7561, pp. 203–207. Springer (2012)
26. Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-based function summaries in bounded model checking. In: Proc. HVC. LNCS, vol. 7261, pp. 160–175. Springer (2012)
27. Wonisch, D., Wehrheim, H.: Predicate analysis with block-abstraction memoization. In: Proc. ICFEM. LNCS, vol. 7635, pp. 332–347. Springer (2012)
28. Xie, Y., Aiken, A.: Saturn: A scalable framework for error detection using boolean satisfiability. TOPLAS **29**(3), 16 (2007)


# Domain-Independent Interprocedural Program Analysis using Block-Abstraction Memoization

Dirk Beyer
LMU Munich, Germany

Karlheinz Friedberger
LMU Munich, Germany

## ABSTRACT

Whenever a new software-verification technique is developed, additional effort is necessary to extend the new program analysis to an interprocedural one, such that it supports recursive procedures. We would like to reduce that additional effort. Our contribution is an approach to extend an existing analysis in a modular and domain-independent way to an interprocedural analysis without large changes: We present *interprocedural* block-abstraction memoization (BAM), which is a technique for procedure summarization to analyze (recursive) procedures. For recursive programs, a fix-point algorithm terminates the recursion if every procedure is sufficiently unrolled and summarized to cover the abstract state space.

BAM Interprocedural works for data-flow analysis and for model checking, and is independent from the underlying abstract domain. To witness that our interprocedural analysis is generic and configurable, we defined and evaluated the approach for three completely different abstract domains: predicate abstraction, explicit values, and intervals. The interprocedural BAM-based analysis is implemented in the open-source verification framework CPAchecker. The evaluation shows that the overhead for modularity and domain-independence is not prohibitively large and the analysis is still competitive with other state-of-the-art software-verification tools.

## CCS CONCEPTS

• **Software and its engineering** → **Formal methods**; **Formal software verification**; • **Theory of computation** → **Program verification**; **Verification by model checking**.

## KEYWORDS

Software Verification, Interprocedural Program Analysis, Recursive C Program, Block Abstraction, Procedure Summary

**ACM Reference Format:**
Dirk Beyer and Karlheinz Friedberger. 2020. Domain-Independent Interprocedural Program Analysis using Block-Abstraction Memoization. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3368089.3409718

Funded in part by Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY).
A reproduction package is available on Zenodo [11].


*ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA*

ACM ISBN 978-1-4503-7043-1/20/11.
https://doi.org/10.1145/3368089.3409718

## 1 INTRODUCTION

Software verification has been successfully applied to improve the quality and reliability of computer programs [2, 3, 19, 22, 28, 30, 40]. In the last decades, several algorithms and approaches were developed to perform software model checking for various kinds of C programs. However, only a few verifiers for C support full interprocedural analysis, that is, verification of recursive programs: Only 13 out of 22 tool submissions (17 different tools) in the 2020 competition on software verification [5] participated successfully in the benchmark category of recursive tasks.

A program analysis is called *interprocedural* if procedures are analyzed separately and verification results are merged together from the separate results. The idea is that a program analysis does not depend on long traces through the program, but analyzes procedures independently from each other, such that the result of a procedure's analysis can be used at all call sites with the same context (e. g., with the same abstract arguments). Many verifiers inline called procedures into the calling procedure and verify long traces through a program without any benefit from a modular approach. This not only hinders the reuse of sub-results of the analysis, but also makes it impossible to verify unbounded recursive programs.

We present BAM Interprocedural, a generalization of summary-based interprocedural analysis. The abstract framework is an extension of block-abstraction memoization (BAM) [9, 57] and is currently used to verify reachability properties about programs.

**Example.** We outline how to prove the correctness of the example program in Fig. 1 (illustrated in Fig. 2), which uses two unsigned integer variables $a$ and $b$, and nondeterministically initializes them as input for the recursive procedure *sum*, which returns the sum of its arguments. The program is deemed correct if $error()$ is not called.

This program can not be verified by a default bounded model checker that iteratively unrolls the recursion, because the number of unrollings is unknown. However, using a procedure summary like $ret = m + n$, where $m$ and $n$ are the parameters of procedure *sum* and $ret$ is the return value of the procedure call, would help with the verification. This summary is a valid abstraction for the control-flow for every call of the procedure *sum* and can be applied as a substitution for the initial call in procedure *main* as well as for the recursive call in procedure *sum* itself. For a fully automated analysis, the verification algorithm must come up with this (or some similar) summary and apply it as part of the proof strategy.

This example program requires an abstract domain that tracks relations between variables. Thus, a standard predicate analysis (such as in Sect. 4) is able to infer such predicates (e. g., via CEGAR [27] and interpolation [43]) and can soundly apply procedure summaries for all call-sites of a procedure. In general, our approach works on a domain-independent level and does not depend on SMT-based summaries. The combination of procedure summaries with a fixed-point algorithm computes an over-approximation of the reachable

```
1  void main(void) {
2    uint a = nondet();
3    uint b = nondet();
4    uint s = sum(a, b);
5    if (s != a + b) {
6      error();
7    }
8  }
9
10 uint sum(uint n, uint m) {
11   if (n == 0) {
12     return m;
13   } else {
14     uint tmp = sum(n - 1, m + 1);
15     return tmp;
16   }
17 }
```

**Figure 1: Example program with a recursive procedure *sum***

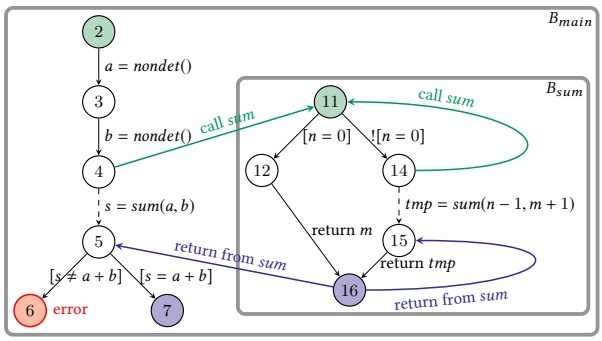



**Figure 2: CFAs for the example program in Fig. 1, with procedure blocks $B_{main}$ and $B_{sum}$**

state space of the recursive procedure. The algorithm first determines a procedure summary for a single unrolling of the procedure, i. e., for all paths through the procedure that are not traversing the recursive call. Using the above mentioned abstract domain, the analysis obtains a summary like $ret = m + n$ in this first step. Then, the algorithm applies the computed procedure summary to the recursive call and explores longer paths through the program and refines the procedure summary until the algorithm cannot explore any new path. For the given example, applying this summary once for the recursive procedure call within the procedure *sum* does not change the summary of the whole procedure *sum*, thus it is sufficient to reach a fixed-point and the analysis can terminate.

**Contribution.** Our contribution consists of three parts:

**(1)** We present a *domain-independent* approach of BAM [57] for a fully interprocedural analysis: every procedure is analyzed separately and the result of a procedure's analysis (an abstraction of the procedure, also known as "procedure summary") is integrated in the analysis of the calling context.

**(2)** A program might contain *unbounded recursion* (e. g., the recursion depth is depending on unknown input). Instead of just cutting off program traces at a predefined depth, our analysis terminates the unrolling of a recursive procedure in a sound way once a fixed point is reached, and does not omit feasible error paths. The fixed-point algorithm iteratively increments the unrolling of the recursion until no new abstract state is reachable. The algorithm is domain-independent, because only coverage checks for abstract states are used, which are already provided by each abstract domain. The overhead is negligible for non-recursive programs.

**(3)** We formally define an *additional domain-specific operator* rebuild in the framework, such that recursive procedures can be handled in every domain. This operator restores eliminated information of the calling context after leaving a recursive call.

**Related Work.** As programs with (recursive) procedures have been analyzed and also verified since decades, many ideas are already available and implemented in some tools. We give a short overview of the tools and the domains they are based on.

***Inlining-Based Analysis.*** A common approach to analyze procedures in bounded model checking is to unroll them up to a certain limit and ignore any deeper recursive calls. Tools like Cbmc [29], Esbmc [36], and Smack [47] implement this approach, which leads to an unsound analysis in combination with recursive procedure calls, because there is no guarantee that the bug is unreachable through further unrolling. Without the user specifying a bound, the model checker might run into an endless unrolling of the recursion. Constant propagation (like in Cbmc) or additional checks can avoid too far unrolling of recursive procedures. Also unbounded frameworks like CPAchecker [15] have several analyses based on different domains [16, 17, 46] that inline procedure calls. Our approach is built on top of them and reuses existing components, such that the amount of changes to a single analysis is minimal.

***Interpolation-Based Summaries.*** Some approaches to verify recursive programs start with a the analysis of single procedures and compute procedure summaries when applying nested function calls. The bounded model checker FunFrog [53, 54] generates interpolation-based [33] procedure summaries to avoid the repeated analysis of procedures. Whale [1] is an extension of Impact [44] and analyzes recursive procedures using two types of formulas in its intra-procedural analysis, namely state- and transition-interpolants, to get summaries. Those approaches separately analyze each procedure until a fixed-point is reached and the procedures (or the representing formulas) are sufficiently refined. UAutomizer uses nested interpolants [38] to compute formulas for procedures depending on the caller's context. Those approaches are bound to an SMT-based domain and the algorithms do not support combinations with other domains.

***Further Domain-Specific Interprocedural Analyses.*** Bebop [4] computes procedure summaries for boolean programs. The application of Bebop however is limited to boolean programs and abstract states are described with binary decision diagrams. Abductor [23] is an interprocedural program verifier that applies the domain of separation logic to prove memory-related safety properties. Additionally, a recursive program can be transformed into a non-recursive one, such that any verification tool without direct support for recursion can be used indirectly to analyze the recursive program. For example, CPArec [26] is a light-weight approach using an external black-box verifier and a fixed-point algorithm that increments the unrolling depth to compute procedure summaries until coverage is reached. This approach is limited to predicate-based verifiers.

***Interprocedural Data-Flow Analysis.*** The above examples are based on symbolic analysis, i. e., depending on BDD-, SAT-, or SMT-based domains, while our proposed approach works for classic data-flow domains as well. Since many years, programs were analyzed in interprocedural manner using several lattice-based domains [31, 50] and with procedure summaries [55]. The classic approach to interprocedural data-flow analysis [21, 41, 48] is restricted to finite-height lattices of domain elements and an operator yielding the join of two domain elements.

BAM Interprocedural works for arbitrary, unlimited abstract domains and different operators for combining elements (depending on the represented data, not only join) or coverage checks for elements (domain-specific comparison).

## 2 BACKGROUND

We describe the program representation as a control-flow automaton and domain-independent reachability analysis based on the concept of configurable program analysis. Afterwards, their application as components in an interprocedural analysis is shown.

### 2.1 Programs

A *program* is represented by a *control-flow automata* (CFA) $A = (L, l_0, G)$ that consists of a set $L$ of program locations, an initial program location $l_0 \in L$, and a set $G \subseteq L \times Ops \times L$ of control-flow edges. An edge models the control-flow operation (from *Ops*) between program locations, for example assignments or assumptions. Figure 2 represents the example program as CFAs. Our presentation uses a simple imperative programming language, which allows only assignments, assume operations, procedure calls and returns, and all variables are integers. The implementation of our tool provides basic support for heap-related data-structures including pointers and arrays, but this article avoids them for simplicity. In general, CPAchecker [15] supports the verification of C programs including pointers and arrays. However, the analysis of recursive procedures for such programs is still under development and a topic of research.

### 2.2 Blocks in a Program

*Blocks* are formally defined as parts of a program: A block $B = (L', G')$ of a CFA $A = (L, l_0, G)$ consists of a set $L' \subseteq L$ of program locations and a set $G' = \{(l_1, op, l_2) \in G \mid l_1, l_2 \in L'\}$ of control-flow edges. We assume that two blocks $B$ and $B'$ are either disjoint ($B.L' \cap B'.L' = \emptyset$) or one block is completely nested in the other block ($B.L' \subset B'.L'$). Each block has *input* and *output locations*, which are defined as $In(B) = \{l \in L' \mid (\exists l' : (l', \cdot, l) \in G \wedge l' \notin L') \vee (\nexists l' : (l', \cdot, l) \in G)\}$ and $Out(B) = \{l \in L' \mid (\exists l' : (l, \cdot, l') \in G \wedge l' \notin L') \vee (\nexists l' : (l, \cdot, l') \in G)\}$, respectively. In general, the block size can be freely chosen in our approach. For an interprocedural analysis, we use procedures as blocks, such that a block abstraction represents a procedure summary. In Fig. 2, the blocks $B_{main}$ and $B_{sum}$ represent the two procedures of the program. The input and output locations are marked in color for each block.

### 2.3 CPA and CPA Algorithm

The reachability analysis is based on the concept of configurable program analysis (CPA) [13], which specifies the abstract domain for a program analysis and additional operations.

A CPA $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ consists of an abstract domain $D$, a transfer relation $\rightsquigarrow$, and the operators merge and stop. The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of a set $C$ of concrete states, a semi-lattice $\mathcal{E} = (E, \sqsubseteq)$ 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 $[\![\cdot]\!] : E \to 2^C$ that maps each abstract-domain element to the represented set of concrete states. The transfer relation $\rightsquigarrow \subseteq E \times E$ computes abstract successor states, a transfer relation $\overset{g}{\rightsquigarrow}$ matches the transfer along an edge $g \in G$ of the CFA. The merge operator $\mathsf{merge} : E \times E \to E$ specifies if and how to merge two abstract states when control flow meets. The stop operator $\mathsf{stop} : E \times 2^E \to \mathbb{B}$ determines whether an abstract state is covered by a given set of abstract states. The operators merge and stop can be chosen appropriately to influence the abstraction level of the analysis. Common choices include $\mathsf{merge}^{sep}(e, e') = e'$ (which does not merge abstract states) and $\mathsf{stop}^{sep}(e, R) = (\exists e' \in R : e \sqsubseteq e')$ (which determines coverage by checking whether the given abstract state is less than or equal to any other reachable abstract state according to the semi-lattice).

Given a CPA, we can apply a reachability algorithm (denoted as CPA algorithm in [13]) that explores the abstract state space of a program and computes all reachable abstract states. The stop operator determines the fixed-point criteria, i. e., whether a state has already been discovered before. For the following description, we consider a reachability analysis $CPA(\mathbb{D}, \mathsf{reached}, \mathsf{waitlist})$ using a CPA $\mathbb{D}$ and two sets reached and waitlist of abstract states as input and returning two sets reached′ and waitlist′ of abstract states. The idea is that starting with the given sets of already reached abstract states and a frontier waitlist, the reachability algorithm computes more reachable successors and a new frontier waitlist.

The CPA algorithm can be used as component in a CEGAR-based fixed-point loop [27] to refine the granularity of the current analysis. For simplicity we ignore the precision in this article.

In the following Sect. 3, we describe our interprocedural extension of block-abstraction memoization, and then in Sect. 4 provide an application of the concept to three separate domains: the Callstack-CPA for tracking a call stack of the program, the Value-CPA for tracking variable assignments explicitly, and the Predicate-CPA for handling variable assignments with predicates.

## 3 BAM FOR INTERPROCEDURAL ANALYSIS

Block-Abstraction Memoization (BAM) [57] is a modular and scalable approach for model checking abstract state spaces by leveraging the idea of *divide and conquer*. BAM divides a large program into smaller parts, named *blocks*, and analyzes them separately. The result of a block's analysis is denoted as a *block abstraction*. Block abstractions are stored in a cache. Whenever a larger block depends on a nested block, a block abstraction of the nested block is created during the larger block's analysis. Block abstractions are independent of a concrete domain and work on an abstract level.

There can be several block abstractions for the same block, e. g., depending on different input values of the block.

In the following, we use procedures as blocks. More precisely, a *procedure block* $B_f$ consists of the procedure $f$ itself and all procedures that are (transitively) called from $f$, such that the whole control-flow of nested blocks, including call and return edges, is included in the block $B_f$ (see Fig. 2).

BAM ensures efficiency by using a cache $\mathsf{cache} \subseteq (\mathit{Blocks} \times E) \rightharpoonup (2^E \times 2^E)$ for block abstractions, which maps the initial abstract state for a block to the block abstraction. The block abstraction is defined as the set of reached abstract states and the set of frontier abstract states, which both are computed during the block's analysis.

BAM is defined recursively (independent of any recursion in the analyzed program) and repeatedly (nestedly) applies the reachability analysis. Our implementation of BAM uses a *stack* of pairs $p \in \mathit{Blocks} \times E$ that consists of all currently open analyses referenced by their block of the CFA to be analyzed and an initial abstract state (starting point of the block abstraction).

This section defines BAM Interprocedural. We show that procedure blocks correspond to procedure summaries, describe the problems of analyzing recursive procedures, the necessity of the fixed-point algorithm, and a new operator rebuild.

## 3.1 Operators of BAM

BAM uses two complementing operators $\mathsf{reduce} \subseteq \mathit{Blocks} \times E \rightharpoonup E$ and $\mathsf{expand} \subseteq \mathit{Blocks} \times E \times E \rightharpoonup E$, and an additional operator $\mathsf{rebuild} \subseteq E \times E \times E \rightharpoonup E$, to drop or restore context-based information for each analyzed block. A CPA with these three additional operators is called *CPA with BAM operators*. On an abstract level, the reduce operator performs an abstraction of the given abstract state and the expand operator concretizes an abstract state for a given context. These operators aim towards an interprocedural analysis where each block can be analyzed without knowing its concrete context. How much of this context-independence can be achieved depends on the concrete domain (see Sect. 4 for more details). The implicit benefit of the first two operators is an improvement of the cache-hit-rate. The operator reduce drops unimportant information from an abstract state when entering a block. The resulting abstract state is more abstract and is used as cache key and as initial abstract state for the block's analysis. The importance of some information depends on the wrapped analysis and the available block. For example, variables, predicates, or levels of the call stack that are not accessed inside the entered block, but only depend on the surrounding context, might be good candidates to be removed from the abstract state. The operator expand restores removed information for abstract states when applying the block abstraction in the surrounding context. The operator rebuild avoids collisions of program identifiers (like variables) when returning from a (possibly recursive) procedure scope into its calling context. This operator does not compute an abstraction, but performs simple operations depending on the given abstract domain such as renaming variables, substituting predicates, or updating indices.

With these operators, we now formally define the CPA for BAM.

**Algorithm 1** *fixedPoint*$(B_{main}, l_0, e_0)$

**Input:** block $B_{main}$ with initial program location $l_0$, abstract state $e_0$
**Output:** set of reachable states, which all represent output states of the block $B_{main}$
**Global Variables:** boolean flag fixedpointReached
**Variables:** set blockResult of abstract states

```
1: repeat
2:     fixedpointReached := true;
3:     blockResult := applyBlockAbstraction(B_main, e_0);
4: until fixedpointReached
5: return blockResult;
```

## 3.2 BAM as CPA

For usage with the CPA concept (see Sect. 2.3), BAM itself is formalized as a CPA $\mathbb{BAM} = (D_{\mathbb{BAM}}, \rightsquigarrow_{\mathbb{BAM}}, \mathsf{merge}_{\mathbb{BAM}}, \mathsf{stop}_{\mathbb{BAM}})$. As BAM works on an abstract, domain-independent level, it requires a separate abstract-domain-dependent analysis (like the *value analysis* or *predicate analysis*) to track variables, values, and assignments. This separate component analysis is also defined via the CPA concept (see Sect. 4). For the following definition we denote it as a general (wrapped) *CPA with BAM operators* $\mathbb{W} = (D_{\mathbb{W}}, \rightsquigarrow_{\mathbb{W}}, \mathsf{merge}_{\mathbb{W}}, \mathsf{stop}_{\mathbb{W}}, \mathsf{reduce}_{\mathbb{W}}, \mathsf{expand}_{\mathbb{W}}, \mathsf{rebuild}_{\mathbb{W}})$.

(1) The domain $D_{\mathbb{BAM}}$ is the wrapped domain $D_{\mathbb{W}}$, i. e., BAM simply uses the abstract states of the underlying domain.
(2) The transfer relation includes the transfer $e \rightsquigarrow_{\mathbb{BAM}} e'$ for two abstract states $e$ and $e'$ and a block $B$ if

$$e' \in \begin{cases} \mathit{fixedPoint}(B_{main}, l, e) & \text{if } l = l_0 \text{ and } \mathit{stack} = [] \\ \mathit{applyBlockAbstraction}(B, e) & \text{if } l \in \mathit{In}(B) \\ \{e'' \mid e \rightsquigarrow_{\mathbb{W}} e''\} & \text{if } l \notin \mathit{Out}(B) \end{cases}$$

where $l$ is the program location for $e$ and *stack* is the internal stack of nested blocks during the analysis.
The transfer relation applies one of three possible steps: (1) The fixed-point algorithm Alg. 1 is executed if the current program location is the initial program location $l_0$ and the *stack* is empty. (2) At an input location of a block $B$, i. e., if a new nested block would be entered from a surrounding context, we apply the block abstraction returned from the operation *applyBlockAbstraction* (cf. Alg. 2) for the nested block. (3) For output locations of blocks, there is no succeeding abstract state (in the sub-analysis). For other program locations, the wrapped transfer relation $\rightsquigarrow_{\mathbb{W}}$ is applied.
(3) The merge operator $\mathsf{merge}_{\mathbb{BAM}} = \mathsf{merge}_{\mathbb{W}}$ delegates to the wrapped analysis, i. e., BAM merges whenever the underlying domain merges abstract states.
(4) The termination check $\mathsf{stop}_{\mathbb{BAM}} = \mathsf{stop}_{\mathbb{W}}$ delegates to the wrapped analysis, i. e., the coverage relation between abstract states depends on the underlying domain.

The transfer relation $\rightsquigarrow_{\mathbb{BAM}}$ uses the fixed-point algorithm and the computation of block abstractions as explained in the next subsections.

**Algorithm 2** *applyBlockAbstraction*$(B, e_I)$

**Input:** abstract state $e_I$ at a block input location of a block $B$
**Output:** abstract states for the output locations of the analyzed block $B$
**Global Variables:** boolean flag fixedpointReached,
set cache mapping a block and an abstract state to a block abstraction,
sequence stack consisting of pairs of a procedure block and an abstract state
**Variables:** sets reached and waitlist of abstract states for the analysis of the current block

1: $e_i$ := reduce$_{\mathbb{W}}(B, e_I)$;
2: **if** $\exists(B, e_c) \in$ stack : $e_i \sqsubseteq e_c$ **then**
3: **if** cache contains $(B, e_c)$ **then**
4: (reached, ·) := cache$(B, e_c)$;
5: **else**
6: reached := {}
7: fixedpointReached := $false$;
8: **else**
9: **if** cache contains $(B, e_i)$ **then**
10: (reached, waitlist) := cache$(B, e_i)$
11: **else**
12: reached := $\{e_i\}$; waitlist$_r$ := $\{e_i\}$
13: stack.push($(B, e_i)$);
14: (reached, waitlist) := $CPA(\mathbb{W}$, reached, waitlist$)$
15: stack.pop();
16: **if** cache contains $(B, e_i)$ **then**
17: (reached$_{old}$, ·) := cache$(B, e_i)$;
18: **for** $e \in$ reached **do**
19: **if** loc$(e) \in Out(B) \wedge \nexists e' \in$ reached$_{old} : e \sqsubseteq e'$ **then**
20: fixedpointReached := $false$;
21: cache$(B, e_i)$ := (reached, waitlist)
22: $e_{call}$ := getPredecessor$(e_I)$;
23: tmp := {expand$_{\mathbb{W}}(B, e_I, e_o) \mid e_o \in$ reached $\wedge$ loc$(e_o) \in Out(B)$}
24: **return** {rebuild$_{\mathbb{W}}(e_{call}, e_I, e_O) \mid e_O \in$ tmp};

## 3.3 Fixed-Point Algorithm for Unbounded Recursion

An analysis of recursive procedures must handle a possibly unbounded unrolling of the call stack if the information of an abstract state is insufficient to avoid deeper exploration and can not cut off the state space. In our approach, the fixed-point algorithm (*fixedPoint*, Alg. 1) repeatedly analyzes the program using *applyBlockAbstraction* (Alg. 2) from the initial program location onwards. It iteratively increments the number of unrollings and terminates only if coverage was reached for all analyzed procedure calls.

In each iteration of the fixed-point algorithm, we generate an overapproximation of some (more) paths through the recursive procedure (because of the limited unrolling of the recursion) and determine a summary for the currently analyzed procedure block. The termination is decided by a coverage check for the abstract states of the analyzed block summary.

The first iteration of the fixed-point algorithm assumes no valid path through the recursive call. We only explore the non-recursive parts of the program's control flow and skip the recursive call of the procedure. Depending on the abstract domain, the initial summary for the recursive procedure is an empty set of abstract states (Alg. 2, line 6). The block abstraction of a procedure is stored in the cache after returning from the procedure call (Alg. 2, line 21).

$$\frac{\{P\}b = f(a)\{Q\} \quad \vdash \quad \{P \wedge p = a\}B_f\{Q \wedge p = a \wedge b = r\}}{\{P\}b = f(a)\{Q\}}$$

**Figure 3: Hoare's rule for recursion, for a given procedure definition** $f(p)\ \{B_f;\ return\ r;\}$

$$\frac{\{[\![P_e]\!]\}b = f(a)\{[\![Q_e]\!]\} \quad \vdash \quad \{[\![P_e]\!]\}B_f\{[\![Q_e]\!]\}}{\{[\![P_e]\!]\}b = f(a)\{[\![Q_e]\!]\}}$$

**Figure 4: Hoare's rule for recursion (with abstract states)**

In further iterations, we increment the limit of unrollings of the recursive procedure and refine the block abstraction, analyze the program again, starting from the initial program location (and using several intermediate results from the cache), until the procedure summary becomes stable.

## 3.4 Soundness of BAM for Recursion

The fixed-point criteria are based on Hoare's rule for recursion (Fig. 3): if the body of a procedure $f$ satisfies the pre- and post-conditions $P$ and $Q$ (including parameter passing and return values) under the condition that all recursive calls to the procedure $f$ satisfy $P$ and $Q$, then the whole procedure $f$ satisfies $P$ and $Q$. Translated into our model, we use (concretizations of) abstract states as pre- and post-conditions of statements, the procedure and its body corresponds to the procedure's block; Fig. 4 shows the resulting rule. The renaming (or an equivalent operation) of equal identifiers from the recursive call of $f$, which appear in the calling and called procedure $f$, is shifted into a different part of the analysis (see Sect. 3.5 on operator rebuild) and is handled in a sound way.

To determine the fixed-point criteria for termination, Alg. 2 checks the following two properties during the analysis.

Firstly, we try to stop the unrolling of an unbounded recursive procedure by an over-approximating analysis. Thus, before analyzing a new recursive procedure call, we check whether the abstract state at a procedure entry is already covered by any abstract state from the current stack (Alg. 2, line 2). If such a covering abstract state exists, we skip the recursive call and use a procedure summary instead of further exploring the recursive call (Alg. 2, line 3 to 7). The procedure summary consists of either previously computed abstract successor states from the BAM cache or (in case of a cache miss) no successor states at all.

Secondly, because a procedure summary represents only a bounded execution of the called procedure, this approach alone represents only a subset of possible traces in the procedure and might be unsound in cases that require deeper unrolling. To determine if the inserted procedure summaries are sufficient for Hoare's rule of Fig. 4, we check for coverage of the exit state (of the procedure executed with the inserted procedure summary) against the previously computed abstract states (of the procedure summary). This check is performed in lines 18 to 20 of Alg. 2. If the coverage relation is satisfied (for all procedures in the program), then the fixed-point algorithm terminates, because fixedpointReached was never set to $false$ during the iteration. In this case we have found a sound over-approximation of the recursive procedure. Otherwise the fixed-point algorithm continues.

### 3.5 Block-Abstraction Computation with Operators

The operation *applyBlockAbstraction* (cf. Alg. 2) starts with the reduction $\mathsf{reduce}_{\mathbb{W}}(B, e_I)$ of initial abstract state $e_I$ and determines the block abstraction for a block $B$. The block abstraction is either taken from the cache or computed via a separate application of the reachability algorithm (i. e., CPA algorithm). To integrate the block abstraction into a surrounding context, the operators $\mathsf{expand}_{\mathbb{W}}$ and $\mathsf{rebuild}_{\mathbb{W}}$ are applied to each abstract state at the block's output location (lines 23 and 24). The operators reduce and expand abstract or concretize the given abstract state and aim to increase the cache-hit rate of BAM. For an interprocedural approach, they remove and restore (most of) the context-based information of a procedure block.

While the fixed-point algorithm handles over-approximations and refinements of block abstractions, an interesting detail of the implementation remains open: How can we identify and work with symbols, i. e., variable identifiers, across procedure scopes? Identical identifiers for program variables of the same procedure scope are problematic for the analysis of recursive procedures. Due to the modularity of the framework CPAchecker, only a separate call-stack analysis knows about procedure scopes and all other analyses assume unique identifiers across all operations. BAM also tracks information about procedures in its stack, but it does not use this information for detailed analysis of variables and identifiers. Each recursive procedure entry starts a new procedure scope, where the identifiers override existing (valid) identifiers from previous call-stack levels. Entering a procedure and overriding existing identifiers from the calling scope is no problem, because only the most local version of an identifier is available (and visible) in the procedure scope. Leaving the procedure afterwards is more complex, because identifiers are overridden during the procedure's traversal and have to be restored to match the calling context.

A solution like a simple renaming of identifiers is not possible, because each domain has its own way of representing variables. Additionally, each domain must have a strategy for handling scoped variables that allows a consistent use of the cache in BAM.

We solve this problem by using a new operator $\mathsf{rebuild} : E \times E \times E \to E$, and we show how to implement it for different domains. The operator rebuild is applied after analyzing the procedure-exit location (Alg. 2, line 24), i. e., after leaving the block of a (maybe recursive) procedure and after the application of the operator expand. The operator rebuild maps three abstract states (information about the calling context from the procedure call state $e_{call}$, information about the arguments and parameters of the called procedure from the procedure entry state $e_I$, and information about the return value and the block abstraction from the procedure exit state $e_O$) to a new abstract state that is a successor of the procedure call and a valid starting point for the further analysis. The operator rebuild is defined depending on the underlying analysis.

## 4 APPLICATION OF BAM INTERPROCEDURAL TO ABSTRACT DOMAINS

In this section, we describe some component program analyses that can be used by BAM Interprocedural to compute context-independent block abstractions. Using the framework CPAchecker, program analyses are composed of several component CPAs. Component CPAs are defined and implemented for tracking the program counter, the predecessor-successor relationship of the reachability graph, or for combining other CPAs in a composite analysis. Thus, we do not need to specify these aspects when defining a component analysis, but directly specify the component analyses. In the following, we explain an analysis for tracking the call stack and two analyses for analyzing variables and assignments (namely *value analysis* and *predicate analysis*).

***Callstack-CPA.*** The *CPA with BAM operators* for *call-stack analysis* $\mathbb{C} = (D_{\mathbb{C}}, \rightsquigarrow_{\mathbb{C}}, \mathsf{merge}_{\mathbb{C}}, \mathsf{stop}_{\mathbb{C}}, \mathsf{reduce}_{\mathbb{C}}, \mathsf{expand}_{\mathbb{C}}, \mathsf{rebuild}_{\mathbb{C}})$ explicitly tracks the call stack $s = [f_1, \cdots f_n]$ of the program, where $f_1$ to $f_n$ denote procedure scopes for an abstract state $s$.

(1) The domain $D_{\mathbb{C}} = (C, \mathcal{E}_{\mathbb{C}}, [\![\cdot]\!])$ is based on the flat semi-lattice $\mathcal{E}_{\mathbb{C}} = (S \cup \{\top\}, \sqsubseteq)$ for the set $S$ of possible call stacks. The expression $s \sqsubseteq s'$ is fulfilled if $s = s'$ or $s' = \top$, $[\![\top]\!] = C$. For all $s$ in $S$, we have $[\![s]\!] = \{c \in C \mid \mathsf{callstackOf}(c) = s\}$.

(2) The transfer relation $\rightsquigarrow_{\mathbb{C}}$ has the transfer $s \overset{g}{\rightsquigarrow}_{\mathbb{C}} s'$ for CFA edge $g$ and abstract states $s = [f_1, \cdots, f_{n-1}, f_n]$ and $s'$, if
$$s' = \begin{cases} [f_1, \cdots, f_n, f_{n+1}] & \text{if } g \text{ is a procedure call to } f_{n+1} \\ [f_1, \cdots, f_{n-1}] & \text{if } g \text{ is a procedure return from } f_n \\ s & \text{otherwise} \end{cases}$$

(3) The merge operator $\mathsf{merge}_{\mathbb{C}} = \mathsf{merge}^{sep}$ does not combine abstract states.

(4) The termination check $\mathsf{stop}_{\mathbb{C}} = \mathsf{stop}^{sep}$ returns whether the same abstract state was already reached before.

(5) The reduce operator $\mathsf{reduce}_{\mathbb{C}}$ abstracts from a concrete call stack and keeps only the context-relevant suffix. Therefore, it determines the maximal range of procedure scopes of the current block, i. e., procedure scopes that can be popped from the current call stack $[f_1, ..., f_i, ..., f_n]$ during an analysis of the current block. Let the procedure scope $f_i$ be the lowest procedure scope on the stack that is reachable during the block's analysis. Then, the operator keeps only the reachable (most local) procedure scopes from the abstract state: $\mathsf{reduce}_{\mathbb{C}}(B, [f_1, ..., f_i, ..., f_n]) = [f_i, ..., f_n]$.

(6) The expand operator $\mathsf{expand}_{\mathbb{C}}$ restores the removed part of the call stack:
$\mathsf{expand}_{\mathbb{C}}([f_1, ..., f_i, ..., f_n], B, [f_i, ..., f_s]) = [f_1, ..., f_i, ..., f_s]$.

(7) The *call-stack analysis* does not track variables, but the procedure scopes themselves. Thus the rebuild operator is defined as: $\mathsf{rebuild}_{\mathbb{C}}(e_{call}, e_I, e_O) = e_O$.

***Value-CPA.*** The *CPA with BAM operators* for *value analysis* $\mathbb{E} = (D_{\mathbb{E}}, \rightsquigarrow_{\mathbb{E}}, \mathsf{merge}_{\mathbb{E}}, \mathsf{stop}_{\mathbb{E}}, \mathsf{reduce}_{\mathbb{E}}, \mathsf{expand}_{\mathbb{E}}, \mathsf{rebuild}_{\mathbb{E}})$ explicitly tracks the assignments of variables. The CPA is used as described in previous work [12, 17] and extended by BAM operators.

(1) The domain $D_{\mathbb{E}} = (C, \mathcal{E}_{\mathbb{E}}, [\![\cdot]\!])$ is based on the semi-lattice $\mathcal{E}_{\mathbb{C}} = (V, \sqsubseteq_{\mathbb{E}})$ for the set $V = (X \rightharpoonup \mathbb{Z})$ of partial functions that model abstract variable assignments for a set $X$ of variables and the set $\mathbb{Z}$ of integer values. We use $v(x)$ to denote the value of a variable $x \in X$ for an abstract variable assignment $v \in V$, and we use $dom(v)$ to denote the set of variables for which $v$ assigns a value, that is, $dom(v) = \{x \mid (x, \cdot) \in v\}$. The partial order $\sqsubseteq_{\mathbb{E}} \subseteq V \times V$ is defined as: $v \sqsubseteq v'$ if $dom(v') \subseteq dom(v)$ and $v(x) = v'(x)$ is

satisfied for all $x \in dom(v')$. The top element $\top_{\mathbb{E}} \in V$ (least upper bound) denotes the abstract variable assignment with no specific value for any variable: $\top_{\mathbb{E}} = \{\}$. The join operator $\sqcup_{\mathbb{E}} : E \times E \to E$ is based on the partial order and returns the least upper bound of its operands. The concretization function $[\![\cdot]\!] : V \to 2^C$ returns the meaning for an abstract variable assignment.

(2) The transfer relation $\rightsquigarrow_{\mathbb{E}}$ has the transfer $v \overset{g}{\rightsquigarrow}_{\mathbb{E}} v'$ for a CFA edge $g = (\cdot, op, \cdot)$ and two abstract variable assignments $v$ and $v'$, if one of the following conditions is satisfied (given a predicate $p$ and an abstract variable assignment $v$, we define $\phi(p, v) := p \wedge \bigwedge_{x \in dom(v)} x = v(x)$ ):
(a) $op = assume(p)$ and predicate $\phi(p, v)$ is satisfiable, and $v'$ is defined as follows: $(x, c) \in v'$ if $c$ is the only satisfying assignment for variable $x$ of the predicate $\phi(p, v)$, or
(b) $op = (\mathtt{w} := exp)$ and $(x, c) \in v'$ if either ($x \neq \mathtt{w}$ and $(x, c) \in v$) or ($x = \mathtt{w}$ and $c$ is the only satisfying assignment for variable $x'$ of the predicate $\phi(x' = exp, v)$ ).

(3) The merge operator $\mathsf{merge}_{\mathbb{E}} = \mathsf{merge}^{sep}$ does not combine abstract states.

(4) The termination check $\mathsf{stop}_{\mathbb{E}} = \mathsf{stop}^{sep}$ returns whether a covering abstract state was already reached before.

(5) The reduce operator $\mathsf{reduce}_{\mathbb{E}}$ only keeps abstract assignments of variables that are accessed in the block's context: $\mathsf{reduce}_{\mathbb{E}}(B, e_I) = \{(x, c) \in e_I \mid x \text{ used in } B\}$.

(6) The expand operator $\mathsf{expand}_{\mathbb{E}}$ restores the assignments that were removed by $\mathsf{reduce}_{\mathbb{E}}$ from the initial abstract state: $\mathsf{expand}_{\mathbb{E}}(e_I, B, e_o) = \{(x, c) \in e_I \mid x \text{ not used in } B\} \cup e_o$.

(7) For the rebuild operator $\mathsf{rebuild}_{\mathbb{E}}$, we define *global* variables as variables declared in the global scope and the rest as *local* variables, i. e., variables declared in a local procedure scope. After leaving a (recursive) procedure call, the operator $\mathsf{rebuild}_{\mathbb{E}}$ considers local variables from the calling scope, and global variables and the return variable [1] from the exited procedure scope: $\mathsf{rebuild}_{\mathbb{E}}(e_{call}, e_I, e_O) =$
$\{(x, c) \in e_{call} \mid \neg isGlobal(x) \wedge \neg isReturn(x)\} \cup$
$\{(x, c) \in e_O \mid isGlobal(x) \vee isReturn(x)\}$.
Because global variables can be assigned during the procedure's execution, they are not reset to their assigned value from before the procedure's execution; their values are taken from the abstract state $e_O$ at the procedure's exit location.

Note that with these definitions of $\mathsf{reduce}_{\mathbb{E}}$ and $\mathsf{expand}_{\mathbb{E}}$, the *value analysis* of a procedure block is not completely detached from the calling context, because a block abstraction for this domain depends on the input values of variables accessed in the block. For procedure blocks, a block abstraction for a function call can be taken from the BAM cache whenever the function arguments and global variables have identical values.

***Predicate-CPA.*** The *CPA with BAM operators* for *predicate analysis* $\mathbb{P} = (D_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \mathsf{merge}_{\mathbb{P}}, \mathsf{stop}_{\mathbb{P}}, \mathsf{reduce}_{\mathbb{P}}, \mathsf{expand}_{\mathbb{P}}, \mathsf{rebuild}_{\mathbb{P}})$ uses predicates to track variables and their values [8, 57]. For this analysis a set $\mathcal{P}$ of predicates is used, which can be incrementally computed in a CEGAR loop [27] that is applied on top of the CPA algorithm. In this description, we do not go into detail on how to determine useful predicates, but assume that the predicates are already available, e. g., by applying an existing refinement strategy [10, 16]. The refinement procedure of the *predicate analysis* computes interpolants that match the structure of the procedure blocks [38] and allow an interprocedural analysis.

For each block $B$, we partition the set $\mathcal{P}$ of predicates into two disjoint sets $\mathcal{P}_B = \{p \in \mathcal{P} \mid p \text{ relevant for } B\}$ and $\mathcal{P}_{\neg B} = \mathcal{P} \setminus \mathcal{P}_B$. A predicate $p \in \mathcal{P}$ is *relevant for* $B$ if it contain variables that are accessed in the block. The partition $\mathcal{P}_{\neg B}$ contains the rest of $\mathcal{P}$.

(1) The domain $D_{\mathbb{P}} = (C, \mathcal{E}_{\mathbb{P}}, [\![\cdot]\!])$ is based on the set $C$ of concrete states, the lattice $\mathcal{E}_{\mathbb{P}} = (E, \sqsubseteq_{\mathbb{P}})$, and a concretization function $[\![\cdot]\!] : E \to C$. The lattice consists of abstract states $e \in E$ that are tuples $(\psi, l^{\psi}, \varphi) \in (\mathcal{P} \times (L \cup \{l_\top\}) \times \mathcal{P})$. The *abstraction formula* $\psi$ is a boolean combination of predicates from $\mathcal{P}$ and has been computed at the program location $l^{\psi}$. The *path formula* $\varphi$ represents (the disjunction of) all paths from $l^{\psi}$ to the abstract state $e$. The partial order $\sqsubseteq \subseteq E \times E$ is defined for any two abstract states $e_1 = (\psi_1, l^{\psi_1}, \varphi_1)$ and $e_2 = (\psi_2, l^{\psi_2}, \varphi_2)$ as: $e_1 \sqsubseteq e_2$ if $(e2 = \top_{\mathbb{P}}) \vee ((l^{\psi_1} = l^{\psi_2}) \wedge (\psi_1 \wedge \varphi_1 \Rightarrow \psi_2 \wedge \varphi_2))$. The top element is $\top_{\mathbb{P}} = (true, l_\top, true)$. The join operator $\sqcup : E \times E \to E$ is based on the partial order and returns the least upper bound of its operands.

(2) The transfer relation $\rightsquigarrow_{\mathbb{P}}$ has the transfer $e \overset{g}{\rightsquigarrow}_{\mathbb{P}} e'$ for an edge $g = (\cdot, op, l')$ and two abstract states $e = (\psi, l^{\psi}, \varphi)$ and $e' = (\psi', l^{\psi'}, \varphi')$, if
$$(\psi', l^{\psi'}, \varphi') = \begin{cases} (true, l', (SP_{op}(\varphi) \wedge \psi)^{\Pi}) & \text{if } blk(e, g) \\ (\psi, l', SP_{op}(\varphi)) & \text{otherwise} \end{cases},$$
where $SP_{op}(\varphi)$ denotes the strongest post-condition of a given path formula $\varphi$ for an operation $op$. The choice of computing a boolean predicate abstraction depends on the configurable operator $blk$. For our work it returns *true* at least for procedure calls, procedure entries, and procedure exits. The boolean predicate abstraction $(\cdot)^{\Pi}$ computes the strongest boolean combination of predicates from $\mathcal{P}$.

(3) The merge operator $\mathsf{merge}_{\mathbb{P}} : E \times E \to E$ combines the two abstract states $e_1 = (\psi_1, l^{\psi_1}, \varphi_1)$ and $e_2 = (\psi_2, l^{\psi_2}, \varphi_2)$ according to their last abstraction computation: $\mathsf{merge}(e_1, e_2) =$
$$\begin{cases} (\psi_2, l^{\psi_2}, \varphi_1 \vee \varphi_2) & \text{if } (\psi_1 = \psi_2) \wedge (l^{\psi_1} = l^{\psi_2}) \\ e_2 & \text{otherwise} \end{cases}$$

(4) The termination check $\mathsf{stop}_{\mathbb{P}} = \mathsf{stop}_{sep}$ returns whether a covering abstract state was already reached before.

(5) For an abstract state $e_I = (\psi_I, l^{\psi_I}, true)$ at a block entry, the operator $\mathsf{reduce}_{\mathbb{P}}$ computes the set $\mathcal{P}_{\neg B} := \{p_1, ..., p_i\}$ of predicates that are irrelevant for the block abstraction and removes them from the abstraction formula:
$\mathsf{reduce}_{\mathbb{P}}(B, e_I) = ((\exists p_1, ..., p_i : \psi_I), l^{\psi_I}, true)$. [2]

(6) The operator $\mathsf{expand}_{\mathbb{P}}$ reverts the operation $\mathsf{reduce}_{\mathbb{P}}$, it computes the set $\mathcal{P}_B := \{p_{i+1}, ..., p_n\}$ of predicates, and restores the full set of predicates $\mathcal{P} = \mathcal{P}_{\neg B} \cup \mathcal{P}_B$ for an output state $e_o = (\psi_o, l^{\psi_o}, true)$ as follows: The abstraction formula $\psi_o$

[1] Our implementation introduces an additional variable to capture the return value, such that we are able to reference it here as well.

[2] We represent the abstraction formula $\psi$ in a way that makes it easy to remove elements from $\mathcal{P}$ in an atomic way from an abstraction formula. (We represent $\psi$ as a binary decision diagram (BDD) whose boolean variables represent predicates from $\mathcal{P}$.)

is extended by the remaining part of the initial abstraction formula $\psi_I$:
$\mathsf{expand}_{\mathbb{P}}(e_I, B, e_o) = ((\exists p_{i+1}, ..., p_n : \psi_I) \wedge \psi_o, l^{\psi_o}, true)$.

(7) The operator $\mathsf{rebuild}_{\mathbb{P}}$ is based on the procedure-call state $e_{call} = (\psi_{call}, l^{\psi_{call}}, true)$, the (not reduced) procedure-entry state $e_I = (\psi_I, l^{\psi_I}, true)$, and the expanded procedure-exit state $e_O = (\psi_O, l^{\psi_O}, true)$. The path formula $\varphi_{call}$ represents the CFA edge that is the procedure entry edge between the program locations of the abstract states $e_{call}$ and $e_I$ and represents the encoding of all assignments of the actual arguments to the formal parameter variables. The operator $\mathsf{rebuild}_{\mathbb{P}}$ computes the predicate abstraction for the conjunction of the abstractions before and after the procedure call and the parameter assignment:
$\mathsf{rebuild}_{\mathbb{P}}(B, e_{call}, e_I, e_O) = (\psi_{call} \wedge \varphi_{call} \wedge \psi_O)^{\Pi}$.

***Interval-CPA.*** The *CPA with BAM operators* for *interval analysis* $\mathbb{I} = (D_{\mathbb{I}}, \rightsquigarrow_{\mathbb{I}}, \mathsf{merge}_{\mathbb{I}}, \mathsf{stop}_{\mathbb{I}}, \mathsf{reduce}_{\mathbb{I}}, \mathsf{expand}_{\mathbb{I}}, \mathsf{rebuild}_{\mathbb{I}})$ tracks variables and the range (interval) of their possible assigned values. The *interval analysis* is similar to the *value analysis*, which can be seen as a special case using intervals containing only one value. The coverage relation between intervals is based on the inclusion of intervals (instead of equality of values). We omit the detailed definition here to keep the reader focused on our approach.

## 4.1 Soundness of Reduce and Expand Operator for the Given Domains

For each of the described domains, the soundness criterion of the whole interprocedural analysis is based on the soundness of the CPA algorithm itself (which we assume as basis) as well as on the properties of the specific operators reduce and expand. For a sound analysis, the abstract states that would have been reached without applying a block abstraction (i. e., only applying the wrapped CPA $\mathbb{W}$) need to be a subset of the states reached with an application of the corresponding block abstraction, i. e., using block abstractions can only be less precise than a wrapped analysis, but never cut off a reachable part of the abstract state space.

The transfer relation $\rightsquigarrow_{\mathbb{BAM}}$ for an abstract state $e \in E$ satisfies the relation $\{e' \in E \mid e \rightsquigarrow_{\mathbb{W}} e'\} \subseteq \{e'' \in E \mid e \rightsquigarrow_{\mathbb{BAM}} e''\}$. Based on the definition of $\rightsquigarrow_{\mathbb{BAM}}$ (Sect. 3.2), the interesting case appears when applying a block abstraction. Thus, the concrete implementation of the operators reduce and expand must satisfy the following condition for all blocks $B$: $\{e' \in E \mid e \rightsquigarrow_{\mathbb{W}} e'\} \subseteq \{\mathsf{expand}(e, B, e_o) \in E \mid \mathsf{reduce}(B, e) \rightsquigarrow_{\mathbb{W}} e_o\}$.

For the call-stack analysis, each abstract call-stack state after an application of a block abstraction exactly matches the call-stack state without such a block abstraction. To prove this, just extend each call stack during the block analysis with the removed part $[f_1, ..., f_{i-1}]$ from the reduce operation. For the value analysis (and based on a programming language without pointer handling), the same proof can be applied: Removing assignments from abstract states and restoring them later results in an abstract state that matches the state when not applying a block abstraction computation. A detailed soundness proof for the predicate domain is given in the literature [57]. Removing irrelevant predicates $P_{\neg B}$ and conjuncting those predicates when applying the block abstraction does only make the analysis more imprecise, but does not reduce the reachable abstract state space.

## 4.2 Embedding BAM Interprocedural in CEGAR

The framework CPAchecker defines BAM as a CPA and allows to combine the CPA algorithm with other algorithms, like CEGAR [27], which allows to refine the granularity of the abstract analysis based on information extracted from infeasible program paths. Additional operators for the refinement step in CEGAR are also defined in a domain-independent manner and available in the framework. In our case, the CEGAR algorithm can wrap the CPA algorithm and the analysis of BAM can benefit from this. Whenever BAM finds a property violation, the reachability analysis and the fixed-point algorithm terminates and the surrounding CEGAR algorithm checks the error path for feasibility. If necessary, CEGAR refines the precision, and BAM with the fixed-point algorithm is re-started with the updated precision.

In case of the *predicate analysis*, the refinement procedure computes tree interpolants [20, 38] according to procedure scopes, i. e., for each entered (and exited) procedure scope along an infeasible error path, a new subtree for the tree interpolation problem is constructed. For other analyses, like *value analysis*, the refinement of recursive procedures does not need special handling. In this case, a refinement strategy for sequential error paths [17] is sufficient.

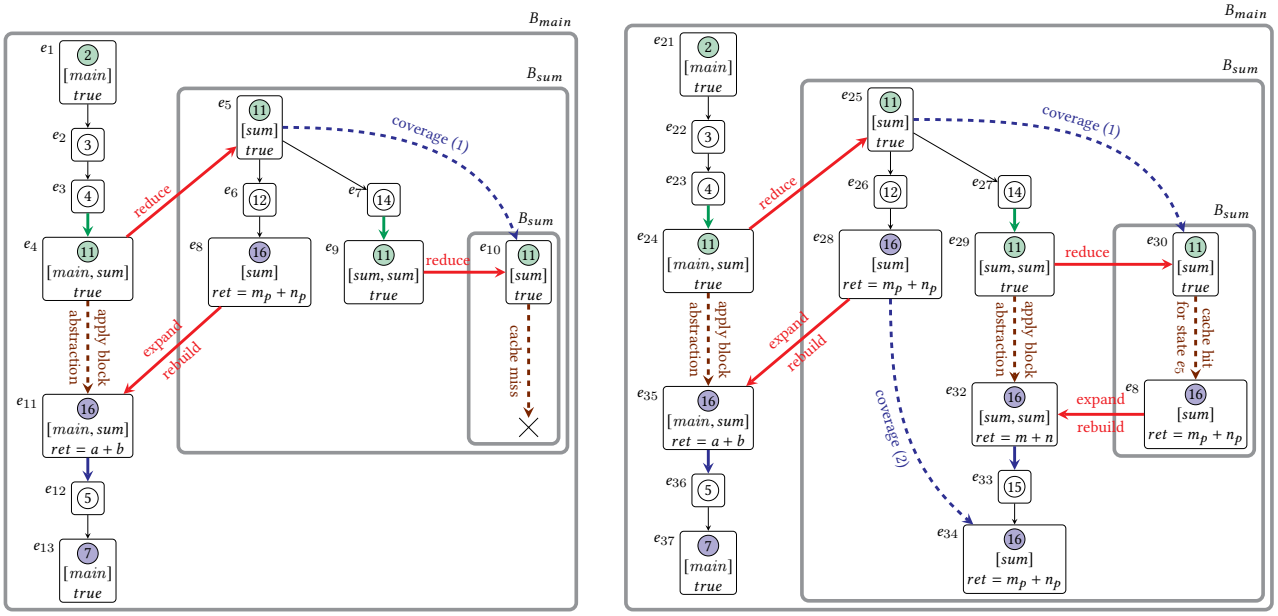
## 4.3 Detailed Description of the Example

In the following, we provide deeper insights for the previously given example program (see Sect. 1) in Fig. 1, to show the control flow of BAM with the fixed-point algorithm when using the *predicate analysis*. We combine the previously defined Callstack-CPA $\mathbb{C}$ and the Predicate-CPA $\mathbb{P}$, i. e., the transfer relation, coverage check, reduce, expand, and rebuild operators are applied in both domains.

Figure 5 shows the abstract states that are reached in the first two iterations of the fixed-point algorithm, which terminates after the second iteration. The labeling of each abstract state consists of the program location (circled number in first line), the call stack (second line), and the abstraction formula of the *predicate analysis* (third line). To keep the figure readable, we dismiss the call stack and abstraction formula whenever there is no change in the abstract state. Outside the upper left corner of each node, we annotate $e_i$, where index $i$ refers to the exploration strategy and control flow of the analysis.

The operators reduce, expand, and rebuild show their effect at the program locations 11 and 16, which are the input and output locations of the procedure block $B_{sum}$. For example, the operator $\mathsf{reduce}_{\mathbb{C}}$ of the *call-stack analysis* removes of all procedure scopes except the most local one from the call stack. The operator $\mathsf{expand}_{\mathbb{C}}$ restores the whole call stack when the analysis leaves the block. The effect of the $\mathsf{rebuild}_{\mathbb{P}}$ at program location 16 will be described below.

*Initialization.* We assume that the initial cache and the stack of BAM are empty and the following set of predicates is defined as precision: $\mathcal{P} := \{ret = m_p + n_p, ret = a + b, m = m_p \wedge n = n_p\}$. The *predicate analysis* uses the symbols $m_p$, $n_p$, and $ret$ to encode parameter assignments at function entry and the return value. Such predicates can be generated via an interpolation procedure from previously found infeasible error paths in the context of CEGAR.




**(a) After first iteration; cache miss leads to second iteration**

**(b) After second iteration; fixed point is reached**

**Figure 5: Graph of reached abstract states after the first two fixed-point iterations**

For simple programs (like this example) they match the expected procedure summary. In general, the analysis might need several iterations of CEGAR to obtain a sufficient precision. In this example, we concentrate on the rebuild operator. All predicates are relevant for the block $B_{sum}$, i. e., $\mathcal{P}_B = \mathcal{P}$, i. e., the reduce and expand operators for *predicate analysis* will keep the abstraction formula unchanged.

*First Iteration.* The result of the first iteration of the fixed-point loop is shown in Fig. 5a. The analysis starts with the initial abstract state $e_1$ at program location 2, entering the main block $B_{main}$ and pushing $e_1$ (as $e_i$ in Alg. 2) onto the BAM stack. The recursive procedure block $B_{sum}$ is analyzed for the first time at the procedure call from program location 4 to program location 11, where BAM starts a new sub-analysis with state $e_4$ (as $e_I$ in Alg. 2) for the block $B_{sum}$. The reduction removes the suffix *main* of the call stack and keeps the abstraction formula *true*. The abstract state $e_5$ (as $e_i$ in Alg. 2) is pushed onto the BAM stack. When the procedure block $B_{sum}$ is entered the second time (procedure call at program location 14 for state $e_9$), the reduced abstract state $e_{10}$ is compared with elements in the BAM stack. The coverage relation (Alg. 2, line 2) is satisfied. BAM has no computed procedure summary in the cache and returns an empty set of reachable abstract states (line 6 of Alg. 2). The flag fixedpointReached is set to *false* in line 7 of Alg. 2. The analysis continues with the exploration of the non-recursive branch of the procedure. When leaving block $B_{sum}$, the block's summary is inserted into the cache, i. e., the block abstraction from the abstract state $e_5$ towards the abstract state $e_8$ (as $e_O$ in Alg. 2) is stored for later usage in the BAM cache. For the predicate analysis, the summary of the block is the abstraction formula $ret = m_p + n_p$, which describes the equality of the sum of the two formal function parameters with the return value.

The rebuild operator $\mathsf{rebuild}(B, e_3, e_4, e_8)$ restores information from the calling context. Using the abstraction formula $\psi_3 := true$, the parameter assignment from the procedure call $\varphi_{call} := (a = n_p \wedge b = m_p)$, and the block summary $\psi_8 := (ret = m_p + n_p)$, the rebuild operator $\mathsf{rebuild}_{\mathbb{P}}$ computes $(\psi_3 \wedge \varphi_{call} \wedge \psi_8)^{\Pi} = (ret = a + b)$. That is, based on the given predicates for $e_{11}$, $\Pi(e_{11}) = \{ret = a + b\}$, the procedure is summarized by $ret = a + b$, which describes the equality of the sum of the two actual function arguments with the return value. We do not describe internals of *predicate abstraction* here, but refer to the literature [16]. No property violation is found along the path until state $e_{13}$, i. e., the branching towards program location 6 is not satisfiable, and the fixed-point computation continues.

*Second Iteration.* The initial steps of the second iteration are similar to the first iteration. After a few steps, the stack consists of the abstract states $e_{21}$ and $e_{25}$. A different control flow appears when the analysis reaches the recursive procedure call again at state $e_{30}$, with a coverage relation for the abstract state $e_{25}$ because it is part of the BAM stack. Now we get a cache hit for the previously computed block abstraction between state $e_5$ and state $e_8$ and apply the procedure summary to skip the recursive procedure call (line 4 of Alg. 2). Using the abstraction formula $\psi_{27} := true$, the parameter assignment from the procedure call $\varphi_{call} := (n = n_p \wedge m = m_p)$, and the block summary $\phi_8 := (ret = m_p + n_p)$, the rebuild operator $\mathsf{rebuild}_{\mathbb{P}}$ computes $(\psi_{27} \wedge \varphi_{call} \wedge \psi_8)^{\Pi} = (ret = m + n)$. When leaving the procedure block, our approach (Alg. 2, line 19) checks for new (not yet covered) abstract states. In this example, state $e_{34}$ is already covered by state $e_{28}$, thus the fixed-point algorithm terminates after this iteration. As the property violation at program location 6 is not reachable, the program is verified.

## 5 EXPERIMENTAL EVALUATION

We evaluate BAM Interprocedural for several domains and show that it is competitive with existing approaches. We divide the evaluation according to three claims. For both claims, we use a benchmark set of non-recursive and recursive programs and provide the effectivity (number of solved problems) and performance (runtime) of our implementation, using several analyses of CPAchecker and other verification tools.

***Claim I: Domain-Independence and Modularity.*** We claim that our interprocedural approach is domain-independent and can be implemented in a modular way as described in Sect. 3, such that the development and integration overhead for an existing analysis in the framework CPAchecker is quite small. To evaluate the claim, we apply the approach to several abstract domains, show that the analysis works, and compare different analyses of CPAchecker against each other.

***Claim II: Effectiveness and Efficiency (Part 1).*** We claim that our approach —despite the modular design— does not cause large performance overheads in an analysis. To evaluate the claim, we compare benchmark results against several state-of-the-art verification tools that are able to verify programs with recursive procedures.

***Claim III: Effectiveness and Efficiency (Part 2).*** We claim that our approach is comparable to intraprocedural analyses within the same framework. To evaluate the claim, we apply different analyses to a larger set of recursive and non-recursive benchmark tasks and compare benchmark results from our interprocedural approach against intraprocedural analyses with and without BAM.

### 5.1 Benchmark Programs and Setup

We use verification tasks from the SV-COMP '20 [5] benchmark set [3], including tasks with and without recursive function calls from categories *Reachsafety-Bitvectors*, *Reachsafety-ControlFlow*, *Reachsafety-Loops*, *Reachsafety-ProductLines*, and *Reachsafety-Recursive*. Most recursive programs are generic and allow to easily scale the programs to deeper recursion; they include recursive algorithms, e. g., Fibonacci, Ackermann, Towers of Hanoi, and McCarthy91. The non-recursive programs use integer arithmetics and avoid heap-related data-structures.

All experiments were performed on machines with a 3.4 GHz Quad Core CPU and 33 GB of RAM. The operating system was Ubuntu 20.04 (64 bit) with Linux 5.4.0. A CPU time limit of 15 min and a memory limit of 15 GB were used, which is the established standard from SV-COMP. Measurements and resource limits were managed by BenchExec [18].

### 5.2 Results and Discussion

***Claim I.*** We implemented our domain-independent approach in CPAchecker for several domains, including *value analysis*, *predicate analysis*, and *interval analysis*. In addition, we evaluated a reduced product [14, 32] of value and predicate analysis. We used CPAchecker in version 1.9, which also participated in SV-COMP '20. CPAchecker was chosen as the implementation platform because it has a configurable and modular design that is easy to extend by new concepts, has a considerable user base, and is well maintained.[4]

**Table 1: Results for the comparison of BAM Interprocedural combined with different abstract domains in CPAchecker on category *Reachsafety-Recursive* of SV-COMP**

| Domain | CPU time (s) | Proofs | Bugs |
|---|---|---|---|
| Value | 924 | 31 | 37 |
| Predicate | 3 440 | 29 | 37 |
| Interval | 849 | 36 | 38 |
| Value + Predicate | 1 690 | 37 | 43 |

**Table 2: Results for the comparison of different verifiers on category *Reachsafety-Recursive* of SV-COMP**

| Verifier | CPU time (s) | Proofs | Bugs |
|---|---|---|---|
| Cbmc | 662 | 32 | 47 |
| CPAchecker (SV-COMP '20) | 2 180 | 37 | 46 |
| Divine | 1 190 | 32 | 42 |
| Esbmc | 941 | 33 | 47 |
| Map2Check | 23 600 | 34 | 37 |
| PeSCo | 3 130 | 37 | 46 |
| Pinaka | 237 | 31 | 31 |
| Symbiotic | 138 | 33 | 45 |
| UAutomizer | 2 160 | 41 | 37 |
| UKojak | 1 010 | 19 | 28 |
| UTaipan | 6 210 | 42 | 37 |
| VeriAbs | 7 630 | 41 | 46 |
| VeriFuzz | 1 960 | 0 | 45 |

Table 1 compares BAM Interprocedural for four domains (one of them being a product), by providing the CPU time (in seconds, with three significant digits) needed by the verifiers for all correctly solved verification tasks and the number of correctly solved tasks, divided into proofs and bugs found in the category *Reachsafety-Recursive* of SV-COMP.

***Claim II.*** We provide the results of state-of-the-art software verifiers, which participated in SV-COMP '20 [5] [5]. We compare 13 verifiers that participated successfully in the category *Reachsafety-Recursive* of SV-COMP. This includes the predicate-based verifiers CPAchecker [35, 56], PeSCo [34, 49] and Ultimate Automizer [37, 39], the bounded model checkers Cbmc [29, 42] and Esbmc [36, 45], the symbolic-execution tool Symbiotic [24, 25], as well as the SMT-based tool Map2Check [51, 52]. The binary archives of all verifiers are publicly available.[6] The data are extracted from the published SV-COMP '20 results [6].

Table 2 provides the sum of CPU time needed by the verifiers for all correctly solved verification tasks, and the number of correctly solved tasks, divided into proofs and bugs found. The configuration used by verifier CPAchecker (SV-COMP '20) combines *value analysis* and *predicate analysis* within our interprocedural approach (same configuration as in the last entry of Table 1), which is automatically selected as the strategy to verify recursive programs [7]. The performance of the tool with our approach (CPAchecker) also shows that

[3] https://github.com/sosy-lab/sv-benchmarks/tree/svcomp20

[4] https://www.openhub.net/p/cpachecker

[5] https://sv-comp.sosy-lab.org/2020/systems.php

[6] https://gitlab.com/sosy-lab/sv-comp/archives-2020/-/tree/svcomp20

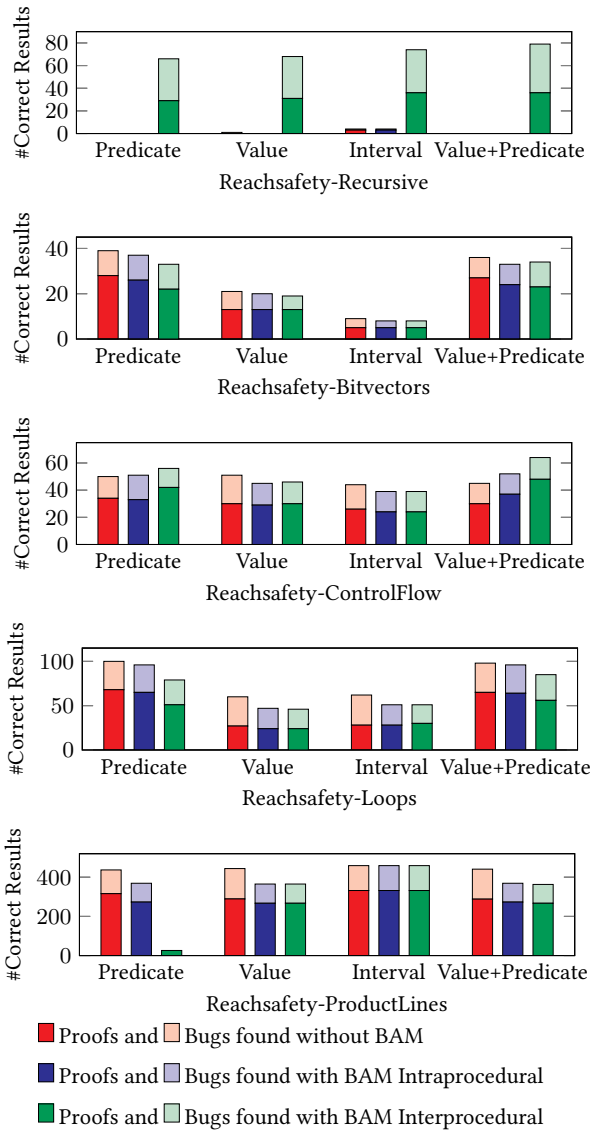



**Figure 6: Results for different benchmark categories for the comparison of different abstract domains without BAM, with BAM Intraprocedural, and with BAM Interprocedural in CPAchecker**

although modular and domain-independent, it is competitive with completely different tools and approaches in terms of *effectiveness* and *efficiency*: BAM Interprocedural solves about as many tasks as the other tools within reasonable CPU time. None of the tools managed to verify all tasks, and there are several tasks in the given benchmark set that could not be solved by any verifier.

***Claim III.*** As CPAchecker is *the* configurable program analysis framework, different domain-independent intraprocedural analyses based on the CPA concept are available, such as the default analysis without BAM and its combination with BAM. Figure 6 compares those algorithms with our new approach of BAM Interprocedural. Each analysis is combined with four different domains (one of them being a product). We provide the number of correctly solved tasks, divided into proofs and bugs found. Each category of SV-COMP '20 is given separately, such that the strengths of the algorithms are visible. In contrast to the existing intraprocedural approaches without and with BAM, the new approach supports the interprocedural analysis of recursive procedures for all three domains separately as well as for a combination of domains and leads to good results in the category *Reachsafety-Recursive*. For all other categories, the results are comparable over all approaches. Only for the predicate domain, the result for the tasks in category *Reachsafety-ProductLines* is worse. The reason for the result in this single category is caused by a valid, but unfitting refinement step (i. e., a suboptimal heuristic in the SMT solver), that causes expensive unrolling of the program. As many tasks in this category are similar, most results are affected. For *value analysis*, *interval analysis*, and also for the analysis based on value and predicate domain together, the new approach performs approximately as good as the existing approaches without or with BAM.

## 6 CONCLUSION

We have presented BAM Interprocedural, a novel approach to interprocedural program analysis. The new approach is **modular** and **domain-independent**, because it is not integrated in a specific program analysis but wraps an existing analysis. In other words, given an arbitrary abstract domain for *intra*-procedural data-flow analysis, we can turn it into an *inter*-procedural analysis without much (a) development work and (b) performance overhead. We have illustrated in detail how to make *predicate analysis* and *value analysis* interprocedural. Our implementation and experiments show that BAM Interprocedural works well for four different program analyses. The new approach supports **recursive** procedures, because it is not bounded to a fixed number of procedure scopes. We showed the effectiveness on the benchmark set of recursive programs from SV-COMP '20: the approach is able to successfully verify recursive procedures. The new approach is **efficient**, because it is integrated into BAM and does not add much overhead on top of the wrapped abstract domain. Compared to other software verifiers, the new implementation is competitive. Due to the modular approach, the effectiveness and efficiency heavily depends on the wrapped program analysis. Our results are promising and there is potential for optimization in our implementation. We plan to specify the operator rebuild for further domains like binary decision diagrams, symbolic memory graphs, or octagons, e. g., to analyze more difficult memory-accesses in recursive programs.

We hope that other researchers and developers of verification tools can benefit from our approach because it separates the concern of making an analysis interprocedural from the actual work on implementing and improving abstract domains.

**Data Availability Statement.** All benchmark tasks for evaluation, configuration files, a ready-to-run version of our implementation, and tables with detailed results are available in our reproduction package [11]. The source code of our extensions to the open-source verification framework CPAchecker [15] is available in the project repository; see https://cpachecker.sosy-lab.org.

## REFERENCES


[1] A. Albarghouthi, A. Gurfinkel, and M. Chechik. 2012. Whale: An Interpolation-Based Algorithm for Inter-procedural Verification. In *Proc. VMCAI (LNCS 7148)*. Springer, 39–55. https://doi.org/10.1007/978-3-642-27940-9_4

[2] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. 2004. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *Proc. IFM (LNCS 2999)*. Springer, 1–20. https://doi.org/10.1007/978-3-540-24756-2_1

[3] T. Ball, V. Levin, and S. K. Rajamani. 2011. A Decade of Software Model Checking with SLAM. *Commun. ACM* 54, 7 (2011), 68–76. https://doi.org/10.1145/1965724.1965743

[4] T. Ball and S. K. Rajamani. 2000. Bebop: A Symbolic Model Checker for Boolean Programs. In *Proc. SPIN (LNCS 1885)*. Springer, 113–130. https://doi.org/10.1007/10722468_7

[5] D. Beyer. 2020. Advances in Automatic Software Verification: SV-COMP 2020. In *Proc. TACAS (2) (LNCS 12079)*. Springer, 347–367. https://doi.org/10.1007/978-3-030-45237-7_21

[6] D. Beyer. 2020. Results of the 9th International Competition on Software Verification (SV-COMP 2020). Zenodo. https://doi.org/10.5281/zenodo.3630205

[7] D. Beyer and M. Dangl. 2018. Strategy Selection for Software Verification Based on Boolean Features: A Simple but Effective Approach. In *Proc. ISoLA (LNCS 11245)*. Springer, 144–159. https://doi.org/10.1007/978-3-030-03421-4_11

[8] D. Beyer, M. Dangl, and P. Wendler. 2018. A Unifying View on SMT-Based Software Verification. *J. Autom. Reasoning* 60, 3 (2018), 299–335. https://doi.org/10.1007/s10817-017-9432-6

[9] D. Beyer and K. Friedberger. 2018. Domain-Independent Multi-threaded Software Model Checking. In *Proc. ASE*. ACM, 634–644. https://doi.org/10.1145/3238147.3238195

[10] D. Beyer and K. Friedberger. 2018. In-Place vs. Copy-on-Write CEGAR Refinement for Block Summarization with Caching. In *Proc. ISoLA (LNCS 11245)*. Springer, 197–215. https://doi.org/10.1007/978-3-030-03421-4_14

[11] D. Beyer and K. Friedberger. 2020. Reproduction Package for Article 'Domain-Independent Interprocedural Program Analysis using Block-Abstraction Memoization' in Proc. ESEC/FSE 2020. Zenodo. https://doi.org/10.5281/zenodo.4024268

[12] D. Beyer, S. Gulwani, and D. Schmidt. 2018. Combining Model Checking and Data-Flow Analysis. In *Handbook of Model Checking*. Springer, 493–540. https://doi.org/10.1007/978-3-319-10575-8_16

[13] D. Beyer, T. A. Henzinger, and G. Théoduloz. 2007. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Proc. CAV (LNCS 4590)*. Springer, 504–518. https://doi.org/10.1007/978-3-540-73368-3_51

[14] D. Beyer, T. A. Henzinger, and G. Théoduloz. 2008. Program Analysis with Dynamic Precision Adjustment. In *Proc. ASE*. IEEE, 29–38. https://doi.org/10.1109/ASE.2008.13

[15] D. Beyer and M. E. Keremoglu. 2011. CPACHECKER: A Tool for Configurable Software Verification. In *Proc. CAV (LNCS 6806)*. Springer, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

[16] D. Beyer, M. E. Keremoglu, and P. Wendler. 2010. Predicate Abstraction with Adjustable-Block Encoding. In *Proc. FMCAD*. FMCAD, 189–197. https://www.sosy-lab.org/research/pub/2010-FMCAD.Predicate_Abstraction_with_Adjustable-Block_Encoding.pdf

[17] D. Beyer and S. Löwe. 2013. Explicit-State Software Model Checking Based on CEGAR and Interpolation. In *Proc. FASE (LNCS 7793)*. Springer, 146–162. https://doi.org/10.1007/978-3-642-37057-1_11

[18] D. Beyer, S. Löwe, and P. Wendler. 2019. Reliable Benchmarking: Requirements and Solutions. *Int. J. Softw. Tools Technol. Transfer* 21, 1 (2019), 1–29. https://doi.org/10.1007/s10009-017-0469-y

[19] D. Beyer and A. K. Petrenko. 2012. Linux Driver Verification. In *Proc. ISoLA (LNCS 7610)*. Springer, 1–6. https://doi.org/10.1007/978-3-642-34032-1_1

[20] R. Blanc, A. Gupta, L. Kovács, and B. Kragl. 2013. Tree Interpolation in Vampire. In *Proc. LPAR (LNCS 8312)*. Springer, 173–181. https://doi.org/10.1007/978-3-642-45221-5_13

[21] O. Burkart and B. Steffen. 1992. Model Checking for Context-Free Processes. In *Proc. CONCUR (LNCS 630)*. Springer, 123–137. https://doi.org/10.1007/BFb0084787

[22] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. 2015. Moving Fast with Software Verification. In *Proc. NFM (LNCS 9058)*. Springer, 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

[23] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *ACM* 58, 6 (2011), 26:1–26:66. https://doi.org/10.1145/2049697.2049700

[24] M. Chalupa, J. Strejcek, and M. Vitovská. 2018. Joint Forces for Memory Safety Checking. In *Proc. SPIN*. Springer, 115–132. https://doi.org/10.1007/978-3-319-94111-0_7

[25] M. Chalupa, M. Vitovská, M. Jonás, J. Slaby, and J. Strejcek. 2017. SYMBIOTIC 4: Beyond Reachability (Competition Contribution). In *Proc. TACAS (LNCS 10206)*. Springer, 385–389. https://doi.org/10.1007/978-3-662-54580-5_28

[26] Y.-F. Chen, C. Hsieh, M.-H. Tsai, B.-Y. Wang, and F. Wang. 2014. Verifying Recursive Programs Using Intraprocedural Analyzers. In *Proc. SAS (LNCS 8723)*. Springer, 118–133. https://doi.org/10.1007/978-3-319-10936-7_8

[27] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50, 5 (2003), 752–794. https://doi.org/10.1145/876638.876643

[28] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. 2018. *Handbook of Model Checking*. Springer. ISBN: 978-3-319-10574-1 https://doi.org/10.1007/978-3-319-10575-8

[29] E. M. Clarke, D. Kröning, and F. Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proc. TACAS (LNCS 2988)*. Springer, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

[30] B. Cook. 2018. Formal Reasoning About the Security of Amazon Web Services. In *Proc. CAV (2) (LNCS 10981)*. Springer, 38–47. https://doi.org/10.1007/978-3-319-96145-3_3

[31] P. Cousot and R. Cousot. 1977. Static Determination of Dynamic Properties of Recursive Procedures. In *Formal Description of Programming Concepts: Proc. of the IFIP Working Conference on Formal Description of Programming Concepts*. North-Holland, 237–278.

[32] P. Cousot and R. Cousot. 1979. Systematic design of program-analysis frameworks. In *Proc. POPL*. ACM, 269–282. https://doi.org/10.1145/567752.567778

[33] W. Craig. 1957. Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem. *J. Symb. Log.* 22, 3 (1957), 250–268. https://doi.org/10.2307/2963593

[34] M. Czech, E. Hüllermeier, M.-C. Jakobs, and H. Wehrheim. 2017. Predicting Rankings of Software Verification Tools. In *Proc. SWAN*. ACM, 23–26. https://doi.org/10.1145/3121257.3121262

[35] M. Dangl, S. Löwe, and P. Wendler. 2015. CPACHECKER with Support for Recursive Programs and Floating-Point Arithmetic (Competition Contribution). In *Proc. TACAS (LNCS 9035)*. Springer, 423–425. https://doi.org/10.1007/978-3-662-46681-0_34

[36] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole. 2018. ESBMC 5.0: An Industrial-Strength C Model Checker. In *Proc. ASE*. ACM, 888–891. https://doi.org/10.1145/3238147.3240481

[37] M. Heizmann, D. Dietsch, J. Leike, B. Musa, and A. Podelski. 2015. ULTIMATE AUTOMIZER with Array Interpolation. In *Proc. TACAS (LNCS 9035)*. Springer, 455–457. https://doi.org/10.1007/978-3-662-46681-0_43

[38] M. Heizmann, J. Hoenicke, and A. Podelski. 2010. Nested interpolants. In *Proc. POPL*. ACM, 471–482. https://doi.org/10.1145/1706299.1706353

[39] M. Heizmann, J. Hoenicke, and A. Podelski. 2013. Software Model Checking for People Who Love Automata. In *Proc. CAV (LNCS 8044)*. Springer, 36–52. https://doi.org/10.1007/978-3-642-39799-8_2

[40] A. V. Khoroshilov, V. S. Mutilin, A. K. Petrenko, and V. Zakharov. 2009. Establishing Linux Driver Verification Process. In *Proc. Ershov Memorial Conference (LNCS 5947)*. Springer, 165–176. https://doi.org/10.1007/978-3-642-11486-1_14

[41] J. Knoop, O. Rüthing, and B. Steffen. 1996. Towards a tool kit for the automatic generation of interprocedural data-flow analyses. *J. Program. Lang.* 4, 4 (1996), 211–246.

[42] D. Kröning and M. Tautschnig. 2014. CBMC: C Bounded Model Checker (Competition Contribution). In *Proc. TACAS (LNCS 8413)*. Springer, 389–391. https://doi.org/10.1007/978-3-642-54862-8_26

[43] K. L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *Proc. CAV (LNCS 2725)*. Springer, 1–13. https://doi.org/10.1007/978-3-540-45069-6_1

[44] K. L. McMillan. 2006. Lazy Abstraction with Interpolants. In *Proc. CAV (LNCS 4144)*. Springer, 123–136. https://doi.org/10.1007/11817963_14

[45] J. Morse, M. Ramalho, L. C. Cordeiro, D. Nicole, and B. Fischer. 2014. ESBMC 1.22 (Competition Contribution). In *Proc. TACAS (LNCS 8413)*. Springer, 405–407. https://doi.org/10.1007/978-3-642-54862-8_31

[46] P. Müller and T. Vojnar. 2014. CPALIEN: Shape Analyzer for CPACHECKER (Competition Contribution). In *Proc. TACAS (LNCS 8413)*. Springer, 395–397. https://doi.org/10.1007/978-3-642-54862-8_28

[47] Z. Rakamarić and M. Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Proc. CAV (LNCS 8559)*. Springer, 106–113. https://doi.org/10.1007/978-3-319-08867-9_7

[48] T. W. Reps, S. Horwitz, and M. Sagiv. 1995. Precise Interprocedural Data-Flow Analysis via Graph Reachability. In *Proc. POPL*. ACM, 49–61. https://doi.org/10.1145/199448.199462

[49] C. Richter and H. Wehrheim. 2019. PESCO: Predicting Sequential Combinations of Verifiers (Competition Contribution). In *Proc. TACAS (3) (LNCS 11429)*. Springer, 229–233. https://doi.org/10.1007/978-3-030-17502-3_19

[50] N. Rinetzky, M. Sagiv, and E. Yahav. 2005. Interprocedural Shape Analysis for Cutpoint-Free Programs. In *Proc. SAS (LNCS 3672)*. Springer, 284–302. https://doi.org/10.1007/11547662_20

[51] H. O. Rocha, R. Barreto, and L. C. Cordeiro. 2016. Hunting Memory Bugs in C Programs with Map2Check (Competition Contribution). In *Proc. TACAS (LNCS 9636)*. Springer, 934–937. https://doi.org/10.1007/978-3-662-49674-9_64

[52] H. O. Rocha, R. S. Barreto, and L. C. Cordeiro. 2015. Memory Management Test-Case Generation of C Programs Using Bounded Model Checking. In *Proc. SEFM (LNCS 9276)*. Springer, 251–267. https://doi.org/10.1007/978-3-319-22969-0_18

[53] O. Sery, G. Fedyukovich, and N. Sharygina. 2011. Interpolation-Based Function Summaries in Bounded Model Checking. In *Proc. HVC (LNCS 7261)*. Springer, 160–175. https://doi.org/10.1007/978-3-642-34188-5_15

[54] O. Sery, G. Fedyukovich, and N. Sharygina. 2015. Function Summarization-Based Bounded Model Checking. In *Validation of Evolving Software*. Springer, 37–53. https://doi.org/10.1007/978-3-319-10623-6_5

[55] M. Sharir and A. Pnueli. 1981. Two approaches to interprocedural data-flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 189–233. ISBN: 978-0-137-29681-1

[56] D. Wonisch. 2012. Block Abstraction Memoization for CPAchecker (Competition Contribution). In *Proc. TACAS (LNCS 7214)*. Springer, 531–533. https://doi.org/10.1007/978-3-642-28756-5_41

[57] D. Wonisch and H. Wehrheim. 2012. Predicate Analysis with Block-Abstraction Memoization. In *Proc. ICFEM (LNCS 7635)*. Springer, 332–347. https://doi.org/10.1007/978-3-642-34281-3_24

# A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAchecker

Dirk Beyer
LMU Munich, Germany

Karlheinz Friedberger
University of Passau, Germany

Verifying multi-threaded programs is becoming more and more important, because of the strong trend to increase the number of processing units per CPU socket. We introduce a new configurable program analysis for verifying multi-threaded programs with a bounded number of threads. We present a simple and yet efficient implementation as component of the existing program-verification framework CPACHECKER. While CPACHECKER is already competitive on a large benchmark set of sequential verification tasks, our extension enhances the overall applicability of the framework. Our implementation of handling multiple threads is orthogonal to the abstract domain of the data-flow analysis, and thus, can be combined with several existing analyses in CPACHECKER, like value analysis, interval analysis, and BDD analysis. The new analysis is modular and can be used, for example, to verify reachability properties as well as to detect deadlocks in the program. This paper includes an evaluation of the benefit of some optimization steps (e.g., changing the iteration order of the reachability algorithm or applying partial-order reduction) as well as the comparison with other state-of-the-art tools for verifying multi-threaded programs.

## 1 Introduction

Program verification has successfully been applied to programs to find errors in applications. There exist many approaches to verify single-threaded programs (cf. SV-COMP for an overview [1]), and several of them are already implemented in the open-source program-verification framework CPACHECKER [4, 10]. For multi-threaded programs a new dimension of complexity has to be taken into account: the verification tool has to efficiently analyze all possible thread interleavings. CPACHECKER did not support the analysis of multi-threaded programs for a long time. Our work focuses on a new, simple configurable program analysis that reuses several existing components of the framework. The approach is sound and can be combined with several steps of optimization to achieve an efficient analysis for multi-threaded programs.

Our analysis is based on a standard state-space exploration using a given control-flow automaton that represents the program. For a program state with several active threads, we compute the succeeding program state for each of those threads, i.e. basically we compute every possible interleaving of the threads. The approach is orthogonal to other data-flow based analyses in CPACHECKER, thus it can be combined with algorithms like CEGAR [7] and analyze an potentially infinite state space.

### 1.0.1 Related Work

A prototypical version of our analysis was already applied for the category of concurrent programs during the SV-COMP'16 [1]. Due to some unsupported features and missing parts of the optimization that where implemented later, the score in this category was low at that time. The experimental results that we report show that the current version of the implementation performs much better.

Just like several other tools [15, 8, 9], we explore possible interleavings of different thread executions and our optimization methods include partial order reduction [12]. In contrast to verification techniques

J. Bouda et al. (Eds.): 11th Doctoral Workshop
on Mathematical and Engineering Methods
in Computer Science (MEMICS 2016)
EPTCS 233, 2016, pp. 61–71, doi:10.4204/EPTCS.233.6

for multi-threaded programs like constraint-based representation [13] that limits the domain to Horn clauses and predicate abstraction or sequentialization [11, 16] that transforms the program on source-code level before starting the analysis, our approach computes the interleaving of threads on-the-fly and is independent from the applied analysis. This makes it possible to integrate our approach easily with data-flow analyses of different abstract domains, such as value analysis [5] and BDD analysis [6].

## 2 Analysis of Multi-Threaded Programs in CPAchecker

The following section provides an overview of some basic concepts and definitions used for our approach. We describe the program representation and the details of our configurable program analysis.

### 2.1 Program Representation

A program is represented by a *control-flow automaton* (CFA) $A = (L, l_0, G)$, which consists of a set $L$ of program locations (modeling the program counter), a set $G \subseteq L \times Ops \times L$ (modeling the control flow with assignment and assumption operations from $Ops$), and an initial program location $l_0$ (entry point of the main function).

Let $V$ be the set of variables in the program. The *concrete data state for a program location* assigns a value to each variable from the set $V$; the set $C$ contains all concrete data states. For every edge $g \in G$, the transition relation is defined by $\xrightarrow{g} \subseteq C \times \{g\} \times C$. The union of all edges defines the complete transfer relation $\rightarrow = \bigcup_{g \in G} \xrightarrow{g}$. If there exists a chain of concrete data states $\langle c_0, c_1, ..., c_n \rangle$ with $\forall c_i$ : there exists a program location $l_i$ for which $c_i$ is a concrete data state and $\forall i : 1 \leq i \leq n \Rightarrow \forall i : 1 \leq i \leq n \Rightarrow \exists g : c_{i-1} \xrightarrow{g} c_i \wedge (l_{i-1}, g, l_i) \in G$, then the state $c_n$ is *reachable* from $c_0$ for $l_0$.

Our analysis is a reachability analysis and unrolls the program lazily [14] into an *abstract reachability graph* (ARG) [2]. The ARG is a directed acyclic graph that consists of abstract states (representing the abstract program state, e.g., including program location and variable assignments) and edges modeling the transfer relation that leads from one abstract state to the next one.

### 2.2 ThreadingCPA

CPAchecker is based on the concept of *configurable program analysis* (CPA) [3]. Thus, different aspects of a program are analyzed by different components (denoted as CPAs). A default analysis in CPAchecker [4] uses the LocationCPA to track the program location (program counter) and the CallstackCPA to track call stacks (function calls and their corresponding return location in the CFA). Thus, for the analysis of sequential programs, each abstract state that is reached during an analysis consists of exactly one program location and one call stack.

For the analysis of multi-threaded programs we have developed a new ThreadingCPA that replaces both the LocationCPA and the CallstackCPA and explores the state space of a multi-threaded program on-the-fly. The benefit of the ThreadingCPA is that it is able to track several program locations (one per thread) together with their call stacks (also one per thread). For simplicity of the definition we ignore the handling of call stacks in the next section. The reader can simply assume that for each program location there is also a call stack. The ThreadingCPA has to handle multiple call stacks (one per thread), whereas the CallstackCPA only handles a single call stack.

The definition of the ThreadingCPA $\mathbb{T} = (D_{\mathbb{T}}, \rightsquigarrow_{\mathbb{T}}, merge_{\mathbb{T}}, stop_{\mathbb{T}})$ follows the structure of a configurable program analysis:

**Domain:** The abstract domain $D_{\mathscr{T}} = (C, \mathscr{T}, [[\cdot]])$ is a triple of the set $C$ of concrete states, the flat semi-lattice $\mathscr{T} = (T, \sqsubseteq, \sqcup, \top)$, and the concretization function $[[\cdot]] : \mathscr{T} \to 2^C$. Let $I$ be the set of all possible thread identifiers, e.g., a set of names used to identify threads in the program. The type of abstract states $T : I \rightharpoonup \mathscr{L}$ consists of all assignments of thread identifiers $t \in I$ to program locations $l \in \mathscr{L} = L \cup \{\top_L\}$. The special program location $\top_L$ represents an unknown program location. The top element $\top \in T$, with $\top(t) = \top_L$ for all $t \in I$, is the abstract state that holds no specific program location for any thread identifier. Each abstract threading state $s \in T$ is represented by the assignments $\{t_1 \mapsto l^{t_1}, t_2 \mapsto l^{t_2}, ...\}$ of thread identifiers to their current program location. The partial order $\sqsubseteq$ induces a semi-lattice for the abstract states. The join operator $\sqcup$ yields the least upper bound of given abstract states. The top element $\top$ of the semi-lattice is defined as $\top = \sqcup T$.

**Merge:** The ThreadingCPA uses the merge operator $merge_{sep}$, which does not combine different elements.

**Stop:** The ThreadingCPA uses the termination operator $stop_{sep}$, which defines coverage only in case of equal abstract states.

**Transfer:** The transfer relation $\rightsquigarrow_{\mathbb{T}}$ determines the syntactic successor for the current state and is based on the transfer relation of the LocationCPA. The implementation is simple: The transfer relation returns all possible successors for all threads that are active in an abstract state, i.e., it applies the transfer relation of the LocationCPA for each active thread. Additionally, thread-management-related operations are included, such that creating or joining threads (when calling *pthread_create* or *pthread_join*) is defined. It is in theory sufficient to only handle these two function calls, because other thread-related function calls do not change the number of threads or the progress of the state-space exploration. The transfer relation $\rightsquigarrow_{\mathbb{T}}$ has the transfer $s \overset{g}{\rightsquigarrow} s'$ for two abstract states $s = \{t_1 \mapsto l^{t_1}, t_2 \mapsto l^{t_2}, ..., t_N \mapsto l^{t_N}\}$ and $s' = \{t_1 \mapsto l'^{t_1}, t_2 \mapsto l'^{t_2}, ..., t_N \mapsto l'^{t_M}\}$ and $g = (l^{t_i}, op, l'^{t_i})$ if

1. the operation $op$ matches the *pthread_create* statement for $t_i$ that is in program location $l^{t_i}$ and creates a new thread $t_{new}$ starting from a CFA node $l_0^{t_{new}} \in L$:
$$s' = s \setminus \{t_i \mapsto l^{t_i}\} \cup \{t_{new} \mapsto l_0^{t_{new}}\} \cup \{t_i \mapsto l'^{t_i}\}$$
i.e., an existing thread $t_i$ matches the program location $l^{t_i}$ and moves along the edge $g$ towards program location $l'$, and the initial program location $l_0^{t_{new}}$ of the new thread $t_{new}$ is added to the current abstract state.
2. the operation $op$ matches the *pthread_join* statement for $t_i$ that is in program location $l^{t_i}$ and waits for a thread $t_{exit}$ to exit, $t_{exit}$ exits at program location $l_E^{t_{exit}}$, and $t_{exit} \mapsto l_E^{t_{exit}} \in s$:
$$s' = s \setminus \{t_i \mapsto l^{t_i}\} \setminus \{t_{exit} \mapsto l_E^{t_{exit}}\} \cup \{t_i \mapsto l'^{t_i}\}$$
i.e., an existing thread $t_i$ matches the program location $l^{t_i}$ and moves along the edge $g$ towards program location $l'^{t_i}$, and the program location $l_E^{t_{exit}}$ of the thread $t_{exit}$ is removed from the current abstract state, if the thread $t_{exit}$ has already been at this program location.
3. otherwise, if the operation $op$ is not related to thread management:
$$s' = s \setminus \{t_i \mapsto l^{t_i}\} \cup \{t_i \mapsto l'^{t_i}\}$$
i.e., thread $t_i$ matches the program location $l^{t_i}$ and moves along the edge towards $l'^{t_i}$.

For a basic analysis for multi-threaded programs the handling of the operations *pthread_create* and *pthread_join* is sufficient. Additional thread management like mutex locks (details in Section 4.3) can be applied on top of this transfer relation. We assume C statements as atomic statements, i.e., interleaving of threads is considered to happen on statement level (matching the encoding of the program as CFA). This might be insufficient for real-world programs, but is good enough for several examples and in theory the CFA could be inflated with read and write operations for memory registers.

```
1  pthread_t id1, id2;
2  int i=1, j=1;
3
4  void main() {
5    pthread_create(&id1, 0, t1, 0);
6    pthread_create(&id2, 0, t2, 0);
7
8    pthread_join(id1, 0);
9    pthread_join(id2, 0);
10
11   assert(j <= 8);
12 }
13
14 void t1() {
15   i+=j;
16   i+=j;
17 }
18
19 void t2() {
20   j+=i;
21   j+=i;
22 }
```

Figure 1: Program with concurrent threads

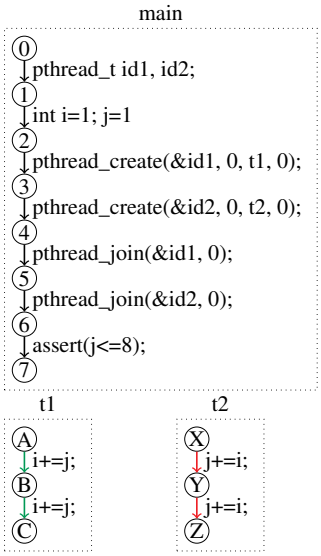



Figure 2: CFA for the functions of the program

## 2.3 Example

The following example applies our new ThreadingCPA to a given program. In contrast to the simplified illustration below, a real-world analysis would combine the ThreadingCPA with another analysis, e.g., to track assignments, such as value analysis or BDD analysis.

The example program (cf. Fig. 1 for the source code) creates two additional threads that change the value of global variables. Afterwards, the main method checks the assignment of a global variable. In this example, the property holds. The program's functions are represented as CFAs in Figure 2. The ThreadingCPA produces the ARG in Fig. 3, where each abstract state is labeled with the indices of the program locations of all active threads.

The analysis starts at entry location $l_0$ of the main function and analyzes all possible interleavings. After reaching the statement *pthread_create*, an additional program location is tracked for the newly created thread, e.g., when reaching program location $l_3$ in the main function, the abstract state is enriched with the initial program location $l_A$ of the newly created thread.

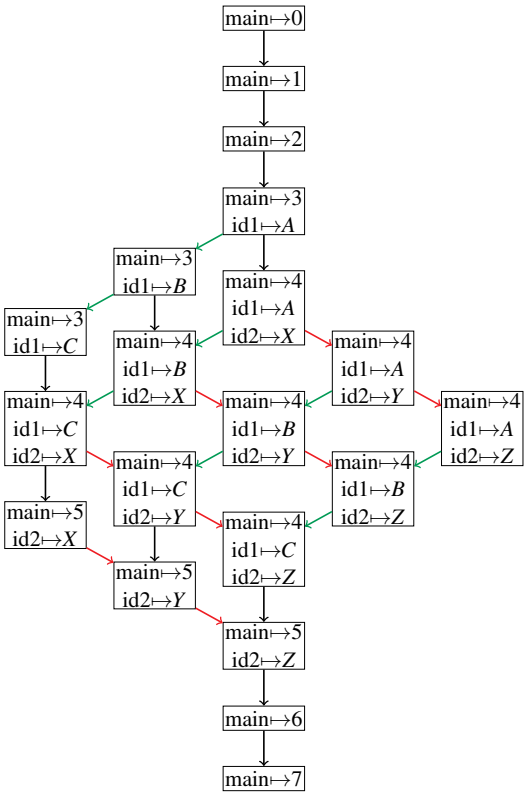



Figure 3: ARG of the interleaved threads of the program

As the ThreadingCPA merges its abstract states when reaching the same program locations via different execution paths, the diamond-like structure in the ARG is the result of interleaved thread-execution of two (or more) threads. When exploring the statement *pthread_join*, the program-exit location of the exiting thread is removed from the abstract state. This is visible in Fig. 3 for each abstract state with an outgoing edge leading from program location $l_4$ towards program location $l_5$, because the program-exit location $l_C$ of the joining thread $t_1$ (identified by id 1) is removed from the abstract state.

# 3 Optimization

The simple definition of the ThreadingCPA allows (and needs) a wide range of optimization to gain competitive efficiency. In the following, we define some approaches and show how fluently they match existing concepts in CPAchecker.

## 3.1 Partitioning of Reached Abstract States

The reachability algorithm [3] has two important operators *merge* and $stop$ that are defined as operations on sets of reached abstract states. These operations can merge abstract states and combine their information into a new abstract state or detect coverage, i.e., an abstract state is implied by another one and thus the exploration can stop at that point. In each iteration of the reachability algorithm, these operators are by default applied to *all* combinations of new explored abstract states and previously reached abstract states. However, applying such an operator to *all* previously reached abstract states is inefficient, because most of the abstract states are irrelevant for a concrete application of these operators. For example, comparing abstract states from different program locations is useless, because there will not be any important relation between them.

*Partitioning* the set of abstract states makes it possible to perform both operations much more efficiently, as only a (small) subset of the previously reached abstract states has to be considered in the computation. This basic optimization is also applied for verifying single-threaded programs. Each partition is identified by a constant key that is based on the program location of the abstract state, as only states from equal program locations are considered for merging or coverage. We extended the existing partitioning of abstract states, such that it uses the tuple of program locations for all threads in an abstract state. This new partitioning can also be combined with partitionings provided by other CPAs.

## 3.2 Waitlist Order

For finding property violations it is often sufficient to only analyze interleavings with a low number of thread interleavings. As the exploration algorithm in CPAchecker analyzes the reachable state space state by state, there exists the possibility to *prioritize abstract states* during the exploration: The abstract states waiting to be analyzed are simply sorted by some criteria. This optimization is a heuristic depending on the internal structure of the analyzed program and the executed analysis. For a bug-free program this heuristic does not bring any benefit. however an existing error path in a faulty program might be found sooner.

The most-often used orderings of abstract states cause the state-space exploration to perform either depth-first search (DFS) or breadth-first search (BFS), i.e., the list of waiting abstract states is ordered in the same manner as abstract states are explored (BFS) or reverse (DFS). For multi-threaded programs, we added a new ordering of this list based on the number of active threads, such that states with fewer active threads are considered first. The new ordering can also be combined with existing orderings, i.e., the

first criteria for ordering is based on the number of active threads, the second criteria uses the exploration order.

### 3.3 Partial-Order Reduction

With multi-threaded programs, the most common form of optimization is *partial-order reduction* (POR) [12, 19, 18]. POR aims to avoid unnecessary interleavings of threads and improves the performance of the analysis by reducing the explored state space. However, its application depends on the property to be verified, because all necessary program paths must remain reachable.

In our case (reachability analysis), we started with a simple separation of program operations (modeled as CFA edges) into *thread-local* and *global* operations. We conservatively apply a static analysis for all program variables and memory accesses, on whether they are declared and used in *global* scope or only *locally* in the context of a thread. Because CPACHECKER uses several dummy operations (e.g., for temporary variables or function returns), a majority of CFA edges is marked as *thread-local*.

If a statement is *thread-local* for a thread, we do not simulate any interleaving after analyzing this operation, but the analysis executes the current thread further, until a global operation (in the same thread) is reached. This behavior is *sound*, because no interaction between threads is possible, due to the definition of *thread-local* operations. Thus, we only need to synchronize all available threads after the next *global* transition.

Our approach can analyze program with loops as well, because we execute both paths, i.e., the loop and the concurrent thread, and none of them disables the other path. Thus, any possible interaction between CFA edges of the loop and other threads is considered. Our approach handles loops implicitly, thus we do not have to actively check for loops, but simply apply the reachability algorithm combined with the described POR technique.

## 4 Extensions

During our work on the analysis of multi-threaded programs, we explored some assumptions in CPACHECKER that need to be considered when integrating such a basic analysis as the ThreadingCPA. We also noticed several features that can also be specified or implemented for the analysis of multi-threaded programs. In the following, we describe the extensions that we have developed in order to use the full potential of the framework.

### 4.1 Cloning for CFAs

CPACHECKER has a modular structure, such that many components can be combined without knowing (and depending on) details about each other. As the analysis of multi-threaded programs should fit into this design, we decided not to modify each analysis that should be combined with our new approach, but use an approach that allows us to re-use as much existing code as possible.

The basic problem with the existing components of CPACHECKER is that many of them rely on knowing only their current function scope, and solely identify a variable by its name combined with the name of the function scope it was declared in. For example, many analyses (including value analysis and BDD analysis) use the identifier $f::x$ for a variable $x$ declared in function $f$. This identifier is used in the internal data structures whenever the variable is used during the program analysis. In a multi-threaded program, the same function $f$ might be called in different threads, such that $f::x$ is not unique for one

variable any more at a certain point in the program's execution. The existing analyses do not know about two variables with the same identifier and would, e.g., assign a wrong value to one of them.

Our solution is simple: We use different function names for each thread by *cloning* the function and inserting the corresponding indexed function name. For a function $f$ we create a clone $f'$ by copying the corresponding CFA nodes from $L$ and edges from $G$, while renaming all appearances of the function's identifier in the clone. Cloning functions causes all function-local variables to be unique for different threads in the later applied analysis, e.g., the identifier $f::x$ is distinct from $f'::x$. An analysis using the identifier does not even have to know whether the function is cloned and can simply assume uniqueness of identifiers for all variables.

### 4.2 Deadlock Detection

A deadlock [17] is defined as an abstract state where two (or more) competing actions wait for each other to finish, and thus neither ever does. CPACHECKER allows the user to define the goal of an analysis by giving a specification in form of an automaton. Detecting deadlocks in the program can be done by giving an observer automaton that monitors the abstract states of the ThreadingCPA and reports deadlocks. This approach is independent of any further analysis and can be combined with, e.g., value analysis or BDD analysis.

### 4.3 Mutex Locks

Mutex locks are commonly used to synchronize threads, e.g., to manage access to shared memory. In our implementation, mutex locks are stored as part of the abstract state of the ThreadingCPA. If a mutex lock is requested along a CFA edge, but not available in the preceding abstract state, the transfer relation does not yield a successive abstract state for the CFA edge.

Additionally, we use mutex locks for more use cases: We simulate atomic sequences of statements and some aspects of partial order reduction as mutex locks in the ThreadingCPA. Entering an atomic sequence requires an *atomic mutex lock*, which is released after leaving the atomic sequence. Consecutive CFA edges containing only thread-local operations (see Section 3.3) are modeled and analyzed as atomic sequence.

## 5 Evaluation

In this section we evaluate different configurations of the ThreadingCPA and compare it with other state-of-the-art tools. The evaluation is performed on machines with a 2.6 GHz Intel Xeon E5-2650 v2 CPU running Ubuntu 16.04 (Linux 4.4.0). Each single verification run is limited to 15 min of run time and 15 GB of memory. The 1016 benchmark tasks are taken from the category of multi-threaded programs at SV-COMP'16 [1]. The tasks are C programs, where reaching a specific function call is considered as property violation. We use CPACHECKER [2] 1.6.1 in revision 23011.

[1] `https://github.com/sosy-lab/sv-benchmarks/releases/tag/svcomp16`
[2] `https://cpachecker.sosy-lab.org/`

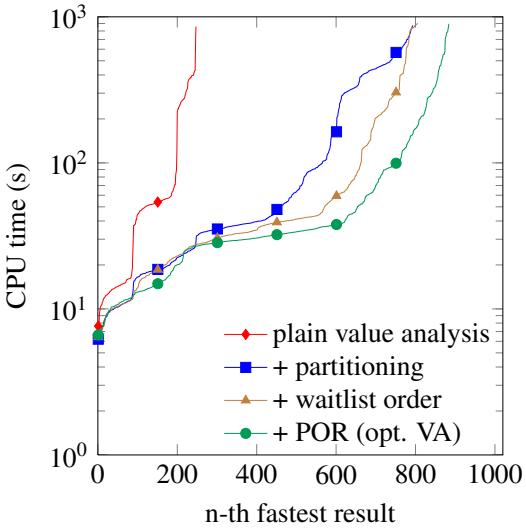



Figure 4: Quantile plot for different configurations of the value analysis, corresponding to step-wise applied optimization steps

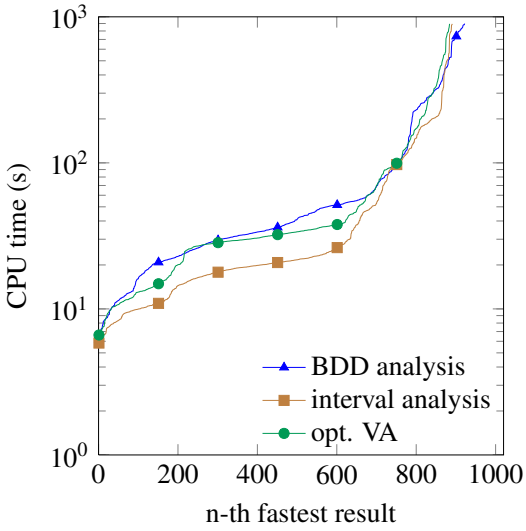



Figure 5: Quantile plot for different abstract domains using the ThreadingCPA within CPACHECKER

## 5.1 Optimization Steps

First, we show the effect of applying each optimization step from Section 3 successively, i.e., on top of the previous optimization. Starting with a plain (non-optimized) configuration of the ThreadingCPA combined with the value analysis, we step-wise apply optimization in form of

- reached-set partitioning (see Section 3.1) based on the abstract states,
- waitlist ordering (see Section 3.2) based on the number of threads, and
- POR (see Section 3.3) based on *local-scope* and *global* statements.

The optimization steps are independent of the value analysis and can also be applied to any other analysis like BDD analysis and interval analysis, where the same benefit will be visible. Figure 4 shows a quantile plot containing the run time of correctly solved verification tasks. The evaluation shows that the verification process benefits from each of the optimization steps. For small tasks that can be verified within a few second, e.g., because of only a few thread interleavings in the program, the benefit of optimization is small. For tasks that need more run time the benefit becomes visible.

We noticed that the heuristic of ordering the waiting abstract states is beneficial in two ways: first, some property violations are found earlier (some property violations need only a small number of interleavings); second, some unsupported operations (like assigning several thread instances to the same thread identifier) are reached earlier and the analysis can abort immediately without wasting time.

Compared to the plain value analysis, partitioning the reached set improves the performance and reduces the run time of the analysis by more than an order of magnitude. Additionally changing the waitlist order improves run time in several cases, mostly for tasks with a property violation. However, in our benchmark this optimization step does not lead to more correctly solved tasks. POR causes a lower number of explored abstract states, and thus the performance increases.

### 5.2 Abstract Domains

Second, we combine the ThreadingCPA with different analyses, such as value analysis, interval analysis, and BDD analysis, which are already implemented in the CPACHECKER framework and are normally used for the analysis of single-threaded programs. We only evaluate the optimized version of each combination. The analyses could also be combined with CEGAR [7], however the current benchmark does not benefit from it, and thus we just execute a reachability algorithm to verify the specification. We show that we can verify the majority of benchmark programs and discuss strengths and weaknesses of the analyses. As all compared analyses use the same framework (parser, algorithm, ...), we expect our evaluation to be fair for all implemented approaches and allow a precise comparison. Figure 5 shows the quantile plot of correct results for the combinations of the ThreadingCPA with other analyses.

The BDD analysis is optimized for BFS in the reachability algorithm, whereas value analysis and interval analysis use DFS as basic order for the list of waiting abstract states during the exploration algorithm (see Section 3.2). Thus, the state-space exploration traverses program locations and thread interleavings in another order and finds the corresponding abstract states in a different order, too. Depending on the verification task, this can result in an in- or decreased performance compared to the value analysis.

### 5.3 Other Tools

Third, we compare the (optimized) value analysis with two other state-of-the-art verification tools, namely CBMC [3] and VVT [4]. Both tools are executed as in the SV-COMP'16 and are chosen, because they do not apply special approaches like sequentialisation, but rely on a similar state-space exploration technique as our approach in CPACHECKER. Figure 6 shows the quantile plot of correct results for CBMC, VVT, and CPACHECKER (using the optimized value analysis). The ThreadingCPA (combined with value analysis) is competitive with the other tools. The plot for CPACHECKER matches the trend of the other tools with only some differences. At the left side of the plot the initial start-up time of a few seconds for CPACHECKER is visible, whereas other tools already solve some of the given instance within this time. Due to the missing support for pointer aliasing and array computations in the value analysis as well as due to our simple kind of POR, CPACHECKER can not solve as many verification tasks as other tools within the time limit.

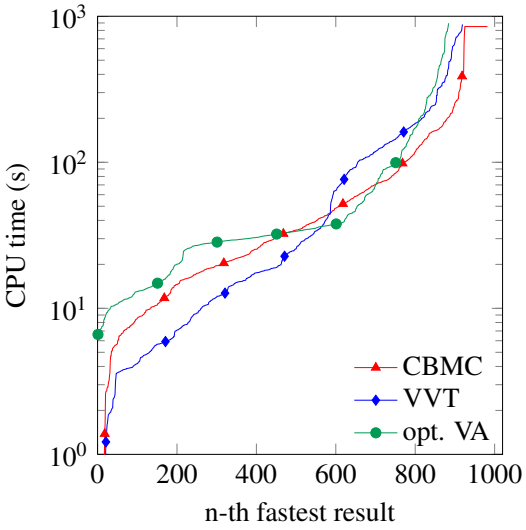



Figure 6: Quantile plot for comparison of other verifiers with support for multi-threaded programs

[3] http://www.cprover.org/cbmc/
[4] https://vvt.forsyte.at/

## 6 Conclusion

This paper presents a basic approach to support the analysis of multi-threaded programs in CPACHECKER. We formally defined a new ThreadingCPA in the framework and demonstrated that several core components can be reused. Re-using existing analyses is possible without any further overhead. Due to our simple approach, there are a few limitations that have to be considered when verifying multi-threaded programs with CPACHECKER. Our approach for partial order reduction is simple and can be extended with more advanced techniques to further reduce the number of explored abstract states. The maximum number of threads is bounded, because of possible conflicts in function names. To avoid naming conflicts, we clone each function's CFA several times before starting the analysis. The number of clones cannot be changed afterwards. If we run out of clones during the analysis and would need more due to a naming conflict, we abort the analysis and report an insufficient number of threads.

As the ThreadingCPA identifies each thread only by the variable it is assigned to, we currently can not analyze more complex thread management such as pointer aliasing for the thread identifier or more complex locking mechanisms. Our framework already contains a mechanism for exchanging information between abstract states on a state-level during the analysis. The analysis of multi-threaded programs could be extended to exchange information about thread management with another analysis capable of such data, such that we could analyze more difficult thread management with the ThreadingCPA.

Possible ideas for optimization have been implemented and evaluated. The evaluation shows that the results of different analyses based on the ThreadingCPA are competitive with other state-of-the-art tools.

## References

[1] D. Beyer (2016): *Reliable and Reproducible Competition Results with* BENCHEXEC *and Witnesses*. In: *Proc. TACAS*, Springer, pp. 887–904, doi:10.1007/978-3-662-49674-9_55. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2016-TACAS.Reliable_and_Reproducible_Competition_Results_with_BenchExec_and_Witnesses.pdf`.

[2] D. Beyer, T. A. Henzinger, R. Jhala & R. Majumdar (2007): *The Software Model Checker* BLAST. *Int. J. Softw. Tools Technol. Transfer* 9(5-6), pp. 505–525, doi:10.1007/s10009-007-0044-z. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2007-STTT.The_Software_Model_Checker_BLAST.pdf`.

[3] D. Beyer, T. A. Henzinger & G. Théoduloz (2007): *Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis*. In: *Proc. CAV*, LNCS 4590, Springer, pp. 504–518, doi:10.1007/978-3-540-73368-3_51. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2007-CAV.Configurable_Software_Verification.pdf`.

[4] D. Beyer & M. E. Keremoglu (2011): CPACHECKER*: A Tool for Configurable Software Verification*. In: *Proc. CAV*, LNCS 6806, Springer, pp. 184–190, doi:10.1007/978-3-642-22110-1_16. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2011-CAV.CPAchecker_A_Tool_for_Configurable_Software_Verification.pdf`.

[5] D. Beyer & S. Löwe (2013): *Explicit-State Software Model Checking Based on CEGAR and Interpolation*. In: *Proc. FASE*, LNCS 7793, Springer, pp. 146–162, doi:10.1007/978-3-642-37057-1_-11. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2013-FASE.Explicit-State_Software_Model_Checking_Based_on_CEGAR_and_Interpolation.pdf`.

[6] D. Beyer & A. Stahlbauer (2013): *BDD-Based Software Model Checking with* CPACHECKER. In: *Proc. MEMICS*, LNCS 7721, Springer, pp. 1–11, doi:10.1007/978-3-642-36046-6_1. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2013-MEMICS.BDD-Based_Software_`

Model_Checking_with_CPAchecker.pdf.

[7] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu & H. Veith (2003): *Counterexample-guided abstraction refinement for symbolic model checking*. J. ACM 50(5), pp. 752–794, doi:10.1145/876638.876643.

[8] E. M. Clarke, D. Kröning, N. Sharygina & K. Yorav (2005): SATABS*: SAT-Based Predicate Abstraction for ANSI-C*. In: *Proc. TACAS*, LNCS 3440, Springer, pp. 570–574, doi:10.1007/978-3-540-31980-1_40.

[9] L. Cordeiro & B. Fischer (2011): *Verifying multi-threaded software using smt-based context-bounded model checking*. In: *Proc. ICSE*, ACM, pp. 331–340, doi:10.1145/1985793.1985839.

[10] Matthias Dangl, Stefan Löwe & Philipp Wendler (2015): CPACHECKER *with Support for Recursive Programs and Floating-Point Arithmetic*. In: *Proc. TACAS*, LNCS 9035, Springer, pp. 423–425, doi:10.1007/978-3-662-46681-0_34.

[11] B. Fischer, O. Inverso & G. Parlato (2013): *CSeq: A Sequentialization Tool for C (Competition Contribution)*. In: *Proc. TACAS*, LNCS 7795, Springer, pp. 616–618, doi:10.1007/978-3-642-36742-7_46.

[12] Patrice Godefroid (1996): *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. LNCS 1032, Springer, doi:10.1007/3-540-60761-7.

[13] A. Gupta, C. Popeea & A. Rybalchenko (2011): *Threader: A Constraint-Based Verifier for Multi-threaded Programs*. In: *CAV*, LNCS 6806, Springer, pp. 412–417, doi:10.1007/978-3-642-22110-1_32.

[14] T. A. Henzinger, R. Jhala, R. Majumdar & G. Sutre (2002): *Lazy abstraction*. In: *Proc. POPL*, ACM, pp. 58–70, doi:10.1145/503272.503279.

[15] G. J. Holzmann (1997): *The* SPIN *Model Checker*. *IEEE Trans. Softw. Eng.* 23(5), pp. 279–295, doi:10.1109/32.588521.

[16] Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre & Gennaro Parlato (2015): *Lazy-CSeq: A Context-Bounded Model Checking Tool for Multi-threaded C-Programs*. In: *Proc. ASE*, ACM, pp. 807–812, doi:10.1109/ASE.2015.108.

[17] Sreekaanth S. Isloor & T. Anthony Marsland (1980): *The Deadlock Problem: An Overview*. *IEEE Computer* 13(9), pp. 58–78, doi:10.1109/MC.1980.1653786.

[18] Doron A. Peled (1993): *All from One, One for All: on Model Checking Using Representatives*. In: *Proc. CAV*, LNCS 697, Springer, pp. 409–423, doi:10.1007/3-540-56922-7_34.

[19] Antti Valmari (1989): *Stubborn sets for reduced state space generation*. In: *Proc. Petri Nets*, LNCS 483, Springer, pp. 491–515, doi:10.1007/3-540-53863-1_36.


# Violation Witnesses and Result Validation for Multi-Threaded Programs

## Implementation and Evaluation with CPAchecker


Dirk Beyer and Karlheinz Friedberger

LMU Munich, Munich, Germany

**Abstract.** Invariants and error traces are important results of a program analysis, and therefore, a standardized exchange format for verification witnesses is used by many program analyzers to store and share those results. This way, information about program traces and variable assignments can be shared across tools, e.g., to validate verification results, or provided to users, e.g., to visualize and explore the results in order to fix bugs or understand the reason for a program's correctness. The standard format for correctness and violation witnesses that was used by SV-COMP for several years was only applicable to sequential (single-threaded) programs. To enable the validation of results for multi-threaded programs, we extend the existing standard exchange format by adding information about thread management and thread interleaving. We contribute a reference implementation of a validator for violation witnesses in the new format, which we implemented as component of the software-verification framework CPAchecker. We experimentally evaluate the format and validator on a large set of violation witnesses. The outcome is promising: several verification tools already produce violation witnesses that help validating the verification results, and our witness validator can re-verify most of the produced witnesses.

**Keywords:** Verification witness · Result validation · Software verification · Proof format · Program analysis · Violation witness · Counterexample · CPAchecker

## 1 Introduction

Reliable and correct software is a basic dependency of today's society and industry. For proving programs correct as well as for finding errors in programs, formal verification is a powerful technique. Given a program and a specification, a software verifier either finds an error path through the program that exposes the specification violation or proves that the specification is satisfied by the program. In most cases, the analysis produces some kind of data that is valuable for the user and can

Replication package available on Zenodo [14].

Funded in part by Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY).



T. Margaria and B. Steffen (Eds.): ISoLA 2020, LNCS 12476, pp. 449–470, 2020.
https://doi.org/10.1007/978-3-030-61362-4_26

be used in further applications. Several tool chains support the direct reuse of verification results [5,6,25]. In general, information about the program analysis can be provided in form of a verification witness, either as correctness witness [10] (e.g., describing invariants from the correctness proof) or as violation witness [11,12] (e.g., representing an abstract counterexample towards a property violation).

The standard witness exchange format was specified and continuously improved by the verification community (especially SV-COMP) over the last years.[1] The specification was first supporting only sequential programs (since SV-COMP 2015 [4,11]), and we later extended it to multi-threaded programs as well (SV-COMP 2018–2020). In this paper, we describe the necessary extensions to the witness format and provide evidence that violation witnesses for concurrent tasks are not only *produced* by many verification tools (in SV-COMP 2020: Cbmc [29], CPAchecker [18], CPAlockator [1], Dartagnan [33], Divine [3], Esbmc [32], Lazy-CSeq [39], PeSCo [31], Ultimate Automizer [37], Ultimate Taipan [35], Yogar-Cbmc [40]), but that most of the violation witnesses for concurrent programs can also be *validated* by our implementation of a validation tool.

**Contributions.** The paper makes the following contributions:

- Extension of the existing violation witness format by additional hints on thread management: (i) thread interleavings are represented using thread-ids at all edges and (ii) thread creation is added to the witness.
- Implementation of an approach for validation of violation witnesses for multi-threaded programs in the verification framework CPAchecker and make the source code available as reference implementation for others.[2]
- Experimental evaluation of the new format and validator on a large number of verification tasks with violation witnesses from several verifiers to show that the approach is effective and helps validating the existence of error traces in multi-threaded programs.
- Availability of all experimental results, including raw data, tables, experiment setup, etc. (see Sect. 6).

**Related Work.** As we extend an existing standardized witness format and validation technology, this work is based on a number of existing ideas, which we outline in the following.

*Verification Artifacts.* Many program-analysis techniques are efficient at discovering proofs or failures. However, it is often difficult to evaluate results, such as program paths towards property violations. Artifacts [24] from verifier executions are valuable for users [2,28,36]. The standard exchange format for verification witnesses [11] is the basis of our work; we describe and extend it in this paper and apply it in our evaluation.

[1] https://github.com/sosy-lab/sv-witnesses
[2] https://cpachecker.sosy-lab.org

*Test Execution and Harnesses.* While it is comparatively simple to create an executable harness for a sequential program [12,27,30,34], the situation for multi-threaded programs is more complex. Simple test cases can not capture the difficulty of nondeterministically interleaved threads and can only be used to heuristically execute a sample of all possible program traces. The scheduling of threads needs to be encoded into the harness in such a way that all statements are interleaved in the correct ordering.

*Sequentialization.* Tools like Lazy-CSeq [38,39] apply sequentialization techniques before verification and can thus provide data about multi-threaded counterexample traces via a sequentialized program. However, the mapping from a sequentialized program (and the found counterexample path in it) back to its multi-threaded origin needs to be supported.

## 2 Background

We provide only a short overview of some basic concepts and definitions that we use to describe our approach, including the program representation, the format for violation witnesses, and the multi-threaded program analysis in CPAchecker.

### 2.1 Program Representation

For presentation, we restrict our programs to a simple imperative programming language that contains only assignments, assumptions, declarations, function calls, and function returns. The language supports simple thread management via the calls of *pthread_ create* and *pthread_ join*, and assumes that each statement in the code is atomic on its own, i.e., uses a strong memory model providing sequential consistency, such that an update of a shared variable is immediately visible to all threads and the verification approach does not need to care about asynchronous memory accesses like simultaneously updating the same memory cell from multiple threads or unit-local caching of values that might happen on hardware level. This is not a theoretical restriction, as each statement could be decomposed into a sequence of reading and writing statements, where each statement involves at most one shared variable. For simplicity and generality, the witnesses ignore further thread-management methods like *mutex locks*, *wait*, and *cancel* operations, as well as interrupts. In violation witnesses such operations do not need to be specified for the validation tool.

The violation witnesses for multi-threaded programs that are produced by the verifiers all support the C programming language as input language and may support a wider range of thread-management operations. We will analyze the quality of those witnesses in the evaluation (Sect. 5).

A program is represented by a *control-flow automaton* (CFA) $(L, l_{init}, G)$, which consists of a set $L$ of program locations (modeling the program counter), a set $G \subseteq L \times Ops \times L$ of control-flow edges (modeling the control flow with assignment and assumption operations as well as declarations and function calls

```
1  int NUM = 4, FIB = 55;
2  int i = 1, j = 1;
3
4  void *t1() {
5    for (int k = 0; k < NUM; k++) {
6      i += j;
7    }
8    pthread_exit(0);
9  }
10
11 void *t2() {
12   for (int k = 0; k < NUM; k++) {
13     j += i;
14   }
15   pthread_exit(0);
16 }
17
18 int main() {
19   pthread_t id1, id2;
20   pthread_create(&id1, 0, t1, 0);
21   pthread_create(&id2, 0, t2, 0);
22   if (i >= FIB || j >= FIB) {
23     __VERIFIER_error();
24   }
25   return 0;
26 }
```

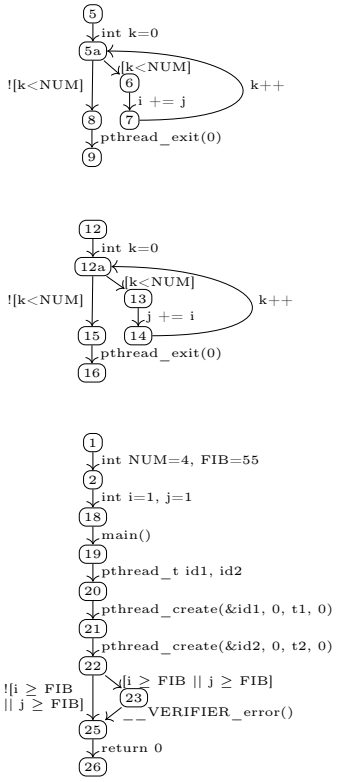



**Fig. 1.** Source code and CFAs for multi-threaded example program, adopted from program https://github.com/sosy-lab/sv-benchmarks/blob/svcomp20/c/pthread/fib_bench-2.c

and returns from *Ops*), and a program-entry location $l_{init} \in L$. A sequence $<g_1, g_2, ..., g_n>$ of CFA edges from G is called *program path* if it starts from the program-entry location (i.e., $g_1 = (l_{init}, \cdot, \cdot)$). As we analyze multi-threaded programs, this sequence consists of potentially interleaved edges from different threads, e.g., there is no need that the end location $l$ of a CFA edge $g_i = (\cdot, \cdot, l)$ is identical with the start location $l'$ of its directly succeeding CFA edge $g_{i+1} = (l', \cdot, \cdot)$, but the next CFA edge along the sequence from the same thread must start with program location $l$. At the program entry and at each thread entry, there is no matching previous program location in a valid program path.

The example in Fig. 1 shows a short multi-threaded program and the corresponding CFAs. The program is build around the Fibonacci sequence, even if the source itself does not directly reveal this. We will later examine this example and find a sequence of operations such that $fib(10) = 55$ was computed (this is modeled as a violation of the specification `G ! call(__VERIFIER_error())` i.e., a call to function `__VERIFIER_error` is not reachable).

### 2.2 Violation Witnesses

Witnesses in software verification are based on the concept of protocol automata [11] that are matched against a CFA for validation. A protocol automaton consists of control states with invariants and edges between control states that represent program transitions. An edge contains a source guard, which restricts the transition to a specific set $S \subseteq G$ of edges from the CFA, and a state-space guard, which restricts the state space by giving additional constraints on variables.

For exporting a violation witness to a file, the protocol automaton is converted into *GraphML* [26], enriched with additional meta-data (like a hash of the analyzed program). When importing a violation witness from a file, the *GraphML* data structure is transformed into a protocol automaton, such that it can be used internally in parallel to any of CPAchecker's program analyses.

### 2.3 Analysis of Multi-Threaded Programs in CPAchecker

CPAchecker is based on the concept of configurable program analysis (CPA) [15,16]. Different concerns of a program are analyzed by different components (denoted as CPAs). To track variables and their assigned values, we can choose from a predicate-abstraction analysis [19], an explicit-value analysis [21], a BDD-based analysis [23], a symbolic execution [20], and several more. For the analysis of program locations in multi-threaded programs, the multi-threading analysis [13] explores the state space, computes possible thread interleavings on-the-fly, and maintains abstract states, where each abstract state consists of several program locations (one per thread) together with their call stacks (also one per thread). Additional optimizations like partial-order reduction are available in the implementation, but not considered here.

To avoid collisions of identifiers during a program analysis, e.g., as it might happen if the same function is called in two different threads at the same time, CPAchecker uses different function names for parallel running threads. If necessary, we use several copies of the CFA for a function of the program, using indexed names. For exporting a violation witness, the indexes are removed, because changed function names are not compatible across different tools. When using an existing violation witness for validating a multi-threaded program, we reintroduce a matching of available thread identifiers in the witness and indexed function copies of the CFAs.

## 3 Detailed Example

In the following, we explain an example step by step. First we start a verifier to verify an example program and produce a witness, and second we start a validator to validate the verification result using the produced witness.

### 3.1 Producing a Violation Witness

The program from Fig. 1 creates two threads `id1` and `id2`, which run in parallel and increase the value of the variables `i` and `j`. If any of the variables `i`

454 D. Beyer and K. Friedberger

| Program Path | Operation Scheduling | | | Variable Values | | | | Line |
|---|---|---|---|---|---|---|---|---|
| | `main` | `id1` | `id2` | `i` | `j` | $k_{t1}$ | $k_{t2}$ | |
| $(1,.,2),(2,.,18),(18,.,19),(19,.,20)$ | i=1, j=1 | | | 1 | 1 | | | 2 |
| $(20,.,21),(5,.,5a)$ | | $k_{t1} = 0$ | | | | 0 | | 5 |
| $(5a,.,6),(6,.,7)$ | | i+=j | | 2 | | | | 6 |
| $(7,.,5a)$ | | $k_{t1}$++ | | | | 1 | | 5 |
| $(21,.,22),(12,.,12a)$ | | | $k_{t2} = 0$ | | | | 0 | 12 |
| $(12a,.,13),(13,.,14)$ | | | j+=i | | 3 | | | 13 |
| $(14,.,12a)$ | | | $k_{t2}$++ | | | | 1 | 12 |
| $(5a,.,6),(6,.,7)$ | | i+=j | | 5 | | | | 6 |
| $(12a,.,13),(13,.,14)$ | | | j+=i | | 8 | | | 13 |
| $(14,.,12a)$ | | | $k_{t2}$++ | | | | 2 | 12 |
| $(7,.,5a)$ | | $k_{t1}$++ | | | | 2 | | 5 |
| $(5a,.,6),(6,.,7)$ | | i+=j | | 13 | | | | 6 |
| $(12a,.,13),(13,.,14)$ | | | j+=i | | 21 | | | 13 |
| $(14,.,12a)$ | | | $k_{t2}$++ | | | | 3 | 12 |
| $(7,.,5a)$ | | $k_{t1}$++ | | | | 3 | | 5 |
| $(5a,.,6),(6,.,7)$ | | i+=j | | 34 | | | | 6 |
| $(12a,.,13),(13,.,14)$ | | | j+=i | | 55 | | | 13 |
| $(22,.,23)$ | j >= FIB | | | | | | | 22 |

**Fig. 2.** Counterexample trace represented by program path, scheduling of operations, data state as variable assignment, and line number as reference

or `j` reaches their limit (which is $fib(10)$), then function `__VERIFIER_error` is reached and a standard verifier can check this by using the specification `G ! call(__VERIFIER_error())` and let it produce a counterexample path. This case can happen if the assignments `i+=j` and `j+=i` in the two threads `id1` and `id2` are executed in alternating order for all iterations of the loops. The rest of the loop statements in both threads, i. e., checking the loop bound, can be executed in arbitrary ordering here and allows a wide range of possible thread interleaving.

The following command line runs CPAchecker as a verifier, configured to use an explicit-value-based analysis for verifying multi-threaded programs:

```
scripts/cpa.sh \
    -outputpath verification \
    -setprop counterexample.export.graphml=witness.graphml \
    -setprop counterexample.export.compressWitness=false \
    -spec config/properties/unreach-call.prp \
    -valueAnalysis-concurrency \
    fib.c
```

This command specifies the directory for all output (including the witness file), the name of the witness file (without compressing it), the specification (which searches for the function call `__VERIFIER_error`), the domain-specific analysis for the verification process, and the subject program.

Violation Witnesses and Result Validation for Multi-Threaded Programs 455


**Fig. 3.** Graphical representation of a violation witness and the available data

The verification process starts the analysis at the program entry and explores the reachable state space. In this example, it finds and reports an error trace as a program path (first column of Fig. 2) and provides the violation witness in Fig. 3, which is written into the file `verification/witness.graphml.gz`. Both, the counterexample trace and the violation witness specify the interleaved thread execution and variable assignments, such that a user or a witness validator can directly follow the path until reaching the property violation in the program. We highlight the information that is relevant for the thread interleaving. The violation witness uses sink nodes for branches or thread interleavings that do not follow the counterexample path. For simplicity, we avoid them in the graphical representation.

Using the explicit-value domain allows us to export detailed data about the counterexample trace, such as assignments for all variables at many program locations. The verification witness is enriched with these assignments, such that the validator can use them as additional constraints.

The information about which thread is executed, and how the interleaving looks like, is important for the user (and also for the validator). In a program with threads created from the same function (that is, with identical line numbers), the thread identifier is the only way to distinguish between different contexts. Therefore, a witness must contain a thread identifier for every transition (edge) in the witness. In this example, the executed threads have different function scopes (`t1` and `t2`) which makes it easier for the reader to find the correct trace towards the property violation.

### 3.2 Validating Results Based on a Violation Witness

In order to validate the information from the witness, the violation witness is matched against the program source code. As the violation witness describes a limited set of paths (best case: exactly one path), the validation process is expected to be efficient and to only analyze a small portion of the reachable state space of the whole program.

The following command line runs CPAchecker as a validator based on the provided violation witness for the multi-threaded program:

```
scripts/cpa.sh \
    -outputpath validation \
    -spec config/properties/unreach-call.prp \
    -witnessValidation \
    -witness verification/witness.graphml \
    fib.c
```

This command specifies the directory for all output (including the newly generated witness file), the specification (which searches for the function call `__VERIFIER_error`), the validation analysis that will select the strategy to analyze multi-threaded programs, the witness that will be used for the validation (as second, parallel specification), and the subject program. Figure 4

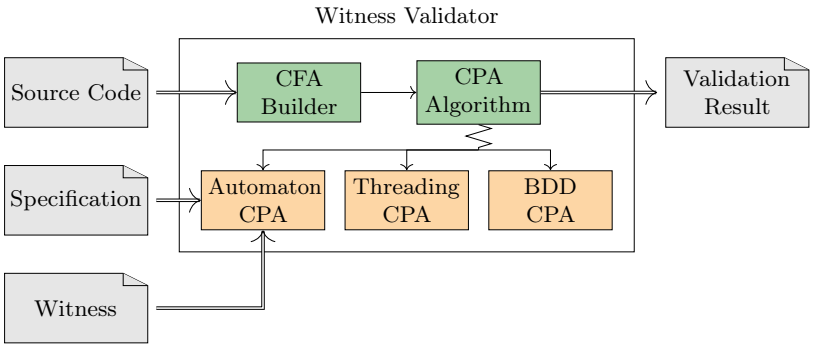



**Fig. 4.** Overview of CPAchecker's control flow for violation witness validation for multi-threaded programs

shows the architecture of CPAchecker for the witness validation for multi-threaded programs. The program is parsed into a CFA and then given to an analysis based on the CPA concept [17]. The property specification and the violation witness are used as protocol automata.

The validation proceeds with the following steps: The witness validator CPAchecker converts the GraphML file (from Fig. 3) into its internal protocol automaton [11], which includes the constraints of the witness. The analysis then runs this automaton in parallel to the default analysis (reduced product) and strengthens the transition relation of the analysis with the additional constraints from the witness. The analysis starts with an initial abstract state built from the program-entry location in the CFA and the entry node in the witness automaton. Then it computes successors for each state and follows a strategy that aims at getting as deep as possible into the witness automaton. This corresponds with strict guidance from the protocol automaton.

By the definition of the witness and the CFA, it is guaranteed that each step through the violation witness matches one or more edges in the program's CFA. The witness structure guides the search towards the property violation in the program. The validator only confirms a property violation from a violation witness, if both the witness automaton and the program location refer to a property violation according to the specification.

For the example, the validation process reports a property violation and confirms the violation witness. The framework reports the validated counterexample trace in form of a new violation witness, which looks quite similar to the existing one. As our validation process uses the BDD-based domain, intermediate steps can be different and more precise than with the previously used explicit-value analysis. However, exporting data from BDDs is more difficult and CPAchecker does not (yet) support it for the witness export.

458 D. Beyer and K. Friedberger

## 4 Violation Witnesses for Multi-threaded Programs

This section gives some details about the extension of the witness format to multi-threaded programs and the implementation of a validator. We used the most obvious way to model traces in multi-threaded programs: specify which thread executes which statement at which point in the trace.

### 4.1 Extending the Existing Format

A violation witness should contain sufficient information about the verification task, such that a validator can efficiently replay the property violation, that is, without re-analyzing the whole state space of the program. This means that for guiding the validator towards a certain property violation, the witness needs to contain sufficient information about all branching choices. While branching points are obvious in sequential programs —just mark all if-then-else statements—, the situation in a multi-threaded program is more complex. The difficulty is to determine the correct ordering of thread interleavings along the counterexample trace. A detailed look provides us insights about the encoding of thread interleavings in CPAchecker: Each program state represents multiple program counters (i.e., one program location per thread) and thus allows the execution of the follower statement from any available thread. We identified only one single information that is critical for the validator to successfully validate a violation witness for a multi-threaded program: a unique *thread identifier* to identify the actual thread that executes a statement given in the witness. Along a violation witness, the thread identifier is required for two different steps:

- Whenever a new thread is started via a control-flow edge calling `pthread_create`, we insert the information `createThread=<ID>`, where `ID` is a new thread identifier for the new thread. Using these hints on *thread creation*, the validator can register a new thread and follow its control flow.
- The thread interleaving is encoded with the *thread identifier* that is given for each statement in the witness. The information `threadId=<ID>` is added to all control-flow edges in the witness, where the thread `<ID>` executes the statement along the control-flow edge.

To keep the witness format as simple as possible, our extension of the witness format consists of only the two above pieces of information (and even those two are optional, i.e., just act as hints for the validator to find the property violation faster). Overall, this allows verification tools that already have support for exporting violation witness and can analyze concurrent programs to directly export violation witnesses for concurrent programs without larger changes to their code base. We considered to include an explicit notion of thread exit or thread join into the set of critical information, but it turned out that none of these actually helped or improved the performance of the validator. In other words, terminating a single thread is unimportant and the validator can automatically infer such information, whenever a single thread reaches the end of its control flow.

**Limitations of the Format.** The current witness format does not support assumptions using thread-local scopes of identifiers, such as *x from thread 1 is larger than x from thread 2*. The validator could in principle overcome this limitation by heuristically choosing which thread is responsible for which identifier. This could make validation slow due to a potentially large overhead. Alternatively, we could extend the assumption format, which are currently plain C statements, with fully qualified names. However, that requires several changes to the syntax (parser and exporter) of the assumptions in both producer and consumer of the witness format. Thus, our validator currently ignores assumptions for which it can not deterministically assign the corresponding thread.

The current witness format does not support quantifiers. For a possibly unbounded number of threads in the program, a correctness witness has to provide information (invariants) over all threads, i.e., uses quantifiers such as *forall threads: property violation can not happen*.

### 4.2 Implementation of the Validator in CPAchecker

CPAchecker transforms a given GraphML-based witness into its internal automaton format, which is then applied along the program analysis to restrict the reachable state space. Additional assumptions over program variables that are given in the witness can be used to strengthen domain-specific transfer relations or cut off the state-space exploration, e. g., if an assumption about a program variable does not satisfy its current assignment.

The validator uses the information from the violation witness for two different features: (1) The state-space exploration is configured to prioritize the search in the direction of the violation witness, i.e., as soon as any control-flow edge from the witness is matching, CPAchecker directly follows that direction. This does not exclude other traces of the program, as they will just be scheduled later in the exploration algorithm. (2) If an assumption is available in the witness, the validator applies *strengthening* and allows to exchange of information between CPAs.

**Matching Thread Identifiers.** The validator needs to combine the information provided in the witness automaton with its own thread model. The important information for multi-threading is provided as an (optional) thread identifier for each single control-flow edge. The validator assumes that the identifier is unique for any particular state in the witness, and we allow to reuse a thread identifier if its previous usage is out of scope, i.e., the corresponding thread has already exited and was joined.

Our internal thread model uses indices to refer to threads in an abstract state. When validating the violation witness, we create a mapping of the thread identifier from the witness to a possible thread index of our own thread model. This allows the validator to be independent from any concrete representation of a thread identifier in the witness.

**Analyses in CPAchecker with Support for Multi-threaded Programs.** The validation for violation witnesses uses the default CPA algorithm [17], which provides an efficient state-space exploration and can be combined with CEGAR.

With the CPA concept, we combine independent analyses (CPAs) that work for different aspects of the program analysis. The automaton analysis handles the matching against the specification automaton and the witness automaton, The threading analysis [13] manages the thread scheduling and interleaving. Additional CPAs like an explicit-value analysis [21], a BDD-based [23], or interval-based analysis allow to reason about assignments of variables.

For validation of violation witnesses, we additionally *strengthen* the abstract threading state with information provided in the witness automaton, in order to track the mapping of thread identifiers and thread indices, and to cut off irrelevant branches in the state space eagerly.

**Limitations of the Validator.** There are some conceptional or implementation-defined limitations of the current implementation of the validator. We discuss these limitations to encourage developers of future validators for multi-threaded programs to improve the approach in our tool, to extend other validators for sequential programs by support for multi-threaded tasks, or to provide new validators.

Based on the requirement to prepare a CFA for each thread of a multi-threaded program, there is a fixed upper bound in the number of threads (default value is 5). The validator ignores traces that use more than the given number of threads, which is an unsound approximation. Note that this is no general limitation of the witness format or the validator. Each concrete error trace for a violation witness has a bounded length and thus can only use a bounded number of thread interleavings. (For example, the number of threads could be added to the metadata of the violation witness and the limit value could be set appropriately.) For the evaluated verification tasks, the default limit was sufficient. If the violation witness is to imprecise and the program allows to create more threads than given in the violation witness, the validator can of course also apply the analysis for more threads. Due to our simple threading analysis, we can only track threads with simple thread-identifier assignments, i.e., where the thread itself is not assigned to an array element or complex pointer structure.

CPAchecker currently supports two domains concretely for analyzing multi-threaded programs, which are explicit-value analysis and BDD-based analysis. The default is to use the BDD-based approach, as it can also handle symbolic values. The validator inherits the limitations of those domains, e.g., it has only limited support for heap-related data structures, such that we need to ignore most array- or pointer-related operations, which can make the validation process imprecise and in some cases even unsound (in case of pointer assignments). This leads to two general cases in which a validator can be wrong: (a) there could be a perfectly valid violation witness but the validator cannot replay it and rejects it due to missing feature support and (b) there could be an invalid violation witness (does not describe a feasible error path) but the validator still finds a different feasible counterexample itself and accepts it due to imprecise information in the witness. There were a few such cases in SV-COMP 2020. The following examples are extracted from the results of SV-COMP 2020 by manual investigation, to give an impression for unsupported features and how they show up in the results[3]:

[3] SV-COMP published all referenced witnesses [9] and verification tasks [8].

- CBMC provides a valid (rather short) violation witness[4] for the task `tls_destructor_worker.yml`[5]. The validator with BDD-based analysis can not confirm this witness due to missing support for `pthread_create_key` and pointer operations.
- ESBMC provides a valid violation witness[6] for the task `race-2_3-container_of.yml`[7], which the validator with BDD-based analysis can not confirm due to missing support for structs.
- YOGAR-CBMC provides a valid violation witness[8] for the task `bigshot_p.yml`[9], for which the validator aborts due to an unexpected assignment of a thread identifier into an array element.

So far, the presented validator is the only validator for multi-threaded programs, and it participated already three years in the competition of software verification (since SV-COMP 2018).

## 5 Experimental Evaluation

We perform an experimental evaluation on violation witnesses for multi-threaded programs to provide qualitative and quantitative insights on how well the result validation based on violation witnesses for multi-threaded programs works.

### 5.1 Evaluation Questions

We split our experimental evaluation into the following evaluation questions:

**Q1:** Which verifiers already support the export of violation witnesses for multi-threaded programs after a successful verification run and what kind of information about the counterexample trace is provided within the witness.
**Q2:** Is the format sufficient and concrete enough for the validator to re-verify the counterexample trace?
**Q3:** Is the validation process faster than the verification process?

[4] https://sv-comp.sosy-lab.org/2020/results/fileByHash/c4a519d36a719304f05e0af3675a0bcf40a7ce4d5000fba784365eed63105ee0.graphml
[5] https://github.com/sosy-lab/sv-benchmarks/blob/svcomp20/c/pthread-divine/tls_destructor_worker.yml
[6] https://sv-comp.sosy-lab.org/2020/results/fileByHash/784befbee140f91b180268f489a6cdce2471ffc6f8578fb0e361c3d2953313d1.graphml
[7] https://github.com/sosy-lab/sv-benchmarks/blob/svcomp20/c/ldv-races/race-2_3-container_of.yml
[8] https://sv-comp.sosy-lab.org/2020/results/fileByHash/f197f473759cc28e4845bcfc6f92af00c0d3ad27e020ee9db1029bfd7c854dba.graphml
[9] https://github.com/sosy-lab/sv-benchmarks/blob/svcomp20/c/pthread/bigshot_p.yml

### 5.2 Benchmark Set

We evaluate the witness format and the validator on a large set of verification tasks, which is taken from the SV-Benchmarks collection [8][10], in the same version as used for SV-COMP 2020. We limit the benchmark set to the subset of verification tasks that exactly matches the category *ConcurrencySafety* in SV-COMP 2020, i.e., multi-threaded programs with a reachability property as specification.

### 5.3 Setup

Our experiments were executed on computers with Intel Xeon E3-1230 v5 CPUs, 3.40 GHz CPU frequency, and 33 GB of RAM. We limited the CPU time to 15 min and the memory to 15 GB.

We evaluated our validator on violation witnesses from SV-COMP [9] that were produced by several different software verifiers. We selected those verifiers that participated in SV-COMP 2020 [7], support violation witnesses (produced more than 100 such witnesses that were confirmed), and have publicly available archives on GitLab[11]. Those verifiers are the following seven: Cbmc, CPA-Seq, Divine, Esbmc, Lazy-CSeq, PeSCo, and Yogar-CBMC. In addition to the witnesses that we took from SV-COMP [9], we also used an updated version of CPAchecker (revision `r33531`) to produce witnesses, where a small extension for the export of violation witnesses was applied (add all beneficial information about thread identifiers to the violation witness and consider more thread interleavings). We include this additional verifier to show that a small and inexpensive extension can lead to a significant improvement of the validation results. The CPU time and memory consumption for each verification run was measured by SV-COMP using BenchExec [22], and the number of nodes and transitions was counted using the GraphML witness files.

Currently, there is only one validator available for violation witnesses of multi-threaded programs, which is the validator explained in Sect. 4.2 and implemented in the CPAchecker framework[2]. We use revision `r33531` for the experiments.

### 5.4 Results and Discussion

*Q1: Verifier Support and Available Information.* All verifiers that we considered in our experiments support (1) the verification of multi-threaded programs and (2) the export of violation witnesses. Some tools include the beneficial information about thread interleaving in the violation witness. That specific feature was already requested in SV-COMP 2018, when the organizers extended the validation of violation witnesses to the category of concurrent tasks. This shows that our extension was already adopted to other verification tools. However, the availability and the quality of the integration differs between the tools.

[10] https://github.com/sosy-lab/sv-benchmarks/tree/svcomp20
[11] https://gitlab.com/sosy-lab/sv-comp/archives-2020/-/tree/svcomp20/2020

**Table 1.** Statistical description of the generated witnesses for the verifiers

| Verifier | Number of states | | | | Number of transitions | | | |
|---|---|---|---|---|---|---|---|---|
| | Median | Mean | Max | Sum | Median | Mean | Max | Sum |
| Cbmc | 6.00 | 6.05 | 10 | 4 790 | 4.00 | 4.05 | 8 | 3 210 |
| CPA-Seq | 48.0 | 48.7 | 662 | 38 700 | 82.0 | 84.4 | 744 | 67 000 |
| CPAchecker (r33531) | 141 | 140 | 1 480 | 112 000 | 207 | 202 | 1 620 | 161 000 |
| Divine | 3.00 | 3.05 | 5 | 1 820 | 2.00 | 2.05 | 4 | 1 220 |
| Esbmc | 3.00 | 5.19 | 30 | 4 140 | 2.00 | 4.19 | 29 | 3 340 |
| Lazy-CSeq | 66.0 | 64.1 | 156 | 52 100 | 64.0 | 62.1 | 154 | 50 500 |
| PeSCo | 51.0 | 49.5 | 662 | 38 600 | 83.0 | 85.9 | 744 | 67 000 |
| Yogar-CBMC | 86.0 | 84.7 | 188 | 68 300 | 84.0 | 82.7 | 186 | 66 700 |

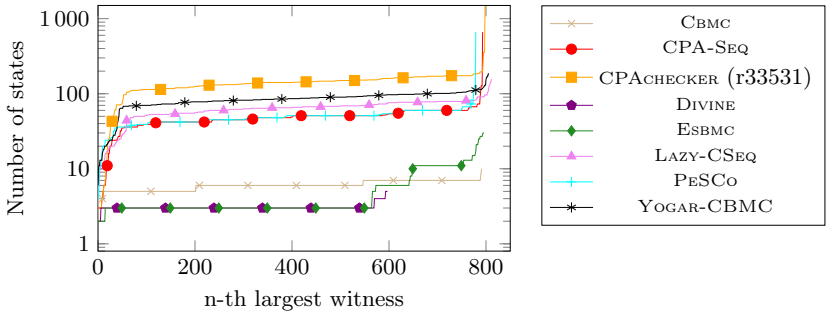
**Table 2.** Properties of the exported violation witnesses

| Verifier | Thread id | Thread creation | All thread interleavings |
|---|---|---|---|
| Cbmc | | ✓ | |
| CPA-Seq | ✓ | ✓ | |
| CPAchecker (r33531) | ✓ | ✓ | ✓ |
| Divine | | | |
| Esbmc | | | |
| Lazy-CSeq | ✓ | ✓ | ✓ |
| PeSCo | ✓ | ✓ | |
| Yogar-CBMC | ✓ | ✓ | ✓ |

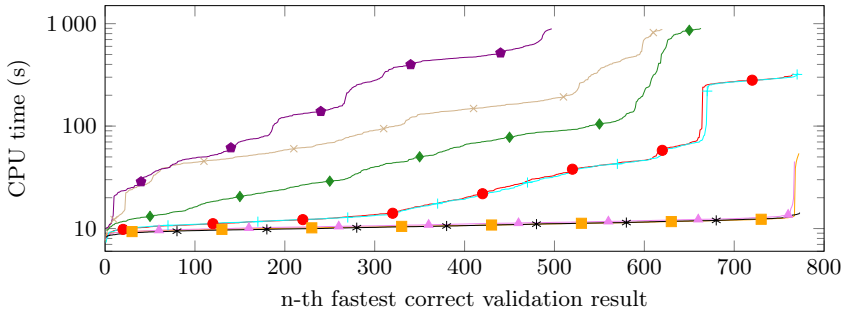
Table 1 gives a statistical overview of the provided violation witnesses and shows how many states and transitions are available in the violation witnesses. Figure 5a shows the distribution of sizes of the violation witnesses for different verifiers. As most tasks have roughly equal difficulty and length of the counterexample trace, the sizes of the violation witnesses are in a certain range. The noticeable difference comes with the tools themselves, i.e., some tools export more details than others.

We also inspected the witnesses for the kind of information they contain. Table 2 shows the different kinds of information available in the witnesses produced by the verifiers. We analyzed whether the violation witnesses contain the *thread id* for every transition, information about *thread creation* for newly started threads during the counterexample trace, and information about thread interleaving. Cbmc, Divine, and Esbmc only export the main thread of the multi-threaded program, which is not suitable for a counterexample trace with interleaving thread statements, because all information about other threads is missing.

*Q2: Validation Results.* The validation results for the produced violation witnesses show whether the information from the violation witness was sufficient to guide the validator towards confirming the given counterexample trace. Overall, the performance of the validation run is determined by two factors: first, how well the violation witness itself guides the state-space exploration and defines the thread




**(a)** Size of witnesses




**(b)** Runtime of validation of witnesses

**Fig. 5.** Quantile plots for violation witnesses from different verifiers

scheduling, and second, how precise the data in the violation witness are. The less information is provided in the violation witness, the more work is left to the validator with its heuristics to recover the error trace. In other words, more precise violation witnesses are often validated faster than less precise witnesses. Figure 5b shows the CPU time of the validator for violation witnesses from different verification tools. Comparing the results with the annotations exported by the tools (Table 2) leads us to a first conclusion: exporting thread interleavings is critical for finding a concrete counterexample path through the program during validation.

As Cbmc, Divine, and Esbmc produce violation witnesses that contain only a minimal set of nodes and transitions, especially only consisting of the main function of the program and ignoring any additional threads, the validation can not follow the given trace sufficiently and performs worse than for other violation witnesses. CPA-Seq and PeSCo use the same underlying analysis, i.e., both tools apply CPAchecker's BDD-based concurrency analyzer with nearly identical options. Thus, they produce nearly identical violation witnesses which also results in similar validation performance.

For the three tools that export thread interleavings in the violation witness, the validation is fast and precise for most of the available verification tasks. Apart from the startup time of the validator (due to starting the Java VM), the

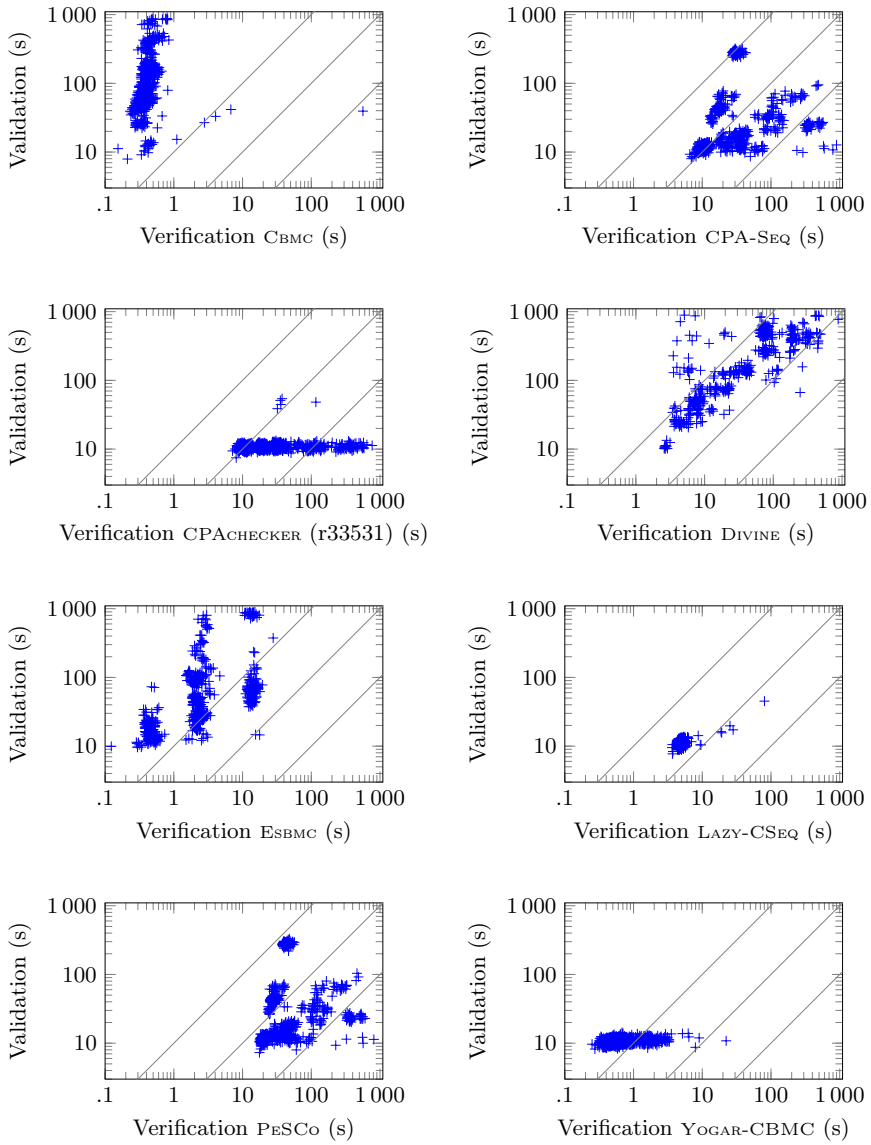



**Fig. 6.** Scatter plot showing the CPU time of the verification process of several tools against the CPU time of the validation process

runtime of the validation itself is negligible. The violation witness guides the validator in the right direction, i.e., all scheduling information is available and nearly no overhead from unimportant program traces appears in the validation process. Only some validation tasks suffer from a high runtime, but these cases also suffer from a rather long and complex to find counterexample trace, such that the violation witness itself contains several thousands of nodes. Note that depending on the verifier, a different path might have been determined, resulting in violation witnesses of different length for each verification task.

*Q3: Performance of Validation Compared to Verification.* Based on the CPU time consumed by the verifiers and the CPU time consumed by the validator, we can compare the performance of the validation with the performance of the verification per verifier. Figure 6 shows several scatter plots, each comparing for a given verifier the CPU times for successful validation runs against the corresponding verification runs. Each data point in the scatter plots represents a verification task that was verified and the resulting violation witness was then successfully validated. The three diagonal lines indicate the factors of 0.1, 1, and 10 between the coordinates.

The overall picture for all scatter plots is as follows: The validator (as part of CPAchecker) is written in Java and has a large startup overhead. This makes it difficult to see a clear performance difference for the small and fast verification tasks. For Cbmc and Esbmc, which are tools with only very imprecise witnesses, the validation usually needs much more CPU time than the actual verification took, i.e., the validator needed much more time to find a counterexample trace matching the rudimentary information in the violation witness. Divine not only has quite imprecise witnesses, but also requires more CPU time for the verification process; thus the difference to the time required for the validation is smaller. For more precise witnesses, as produced by Lazy-CSeq, Yogar-CBMC, and CPAchecker (r33531), the validation process is often faster, or at least requires mostly a nearly constant time (about 10 s).

### 5.5 Threats to Validity

The validity of our experiments is limited by certain choices that we made in the experiment setup.

*External Validity.* The verifiers are all state-of-the-art and seven of them are taken from SV-COMP 2020. We applied the same options and a similar environment that was used in the competition execution, and collected the violation witnesses from the selected verifiers.

There exists only a single validator for multi-threaded violation witnesses, and it might be possible that our results (sufficient information in the witness format) does not apply to other, future validators for multi-threaded programs. We based our validator on the verification framework CPAchecker, because mechanisms for witness export and validation was already integrated. The configuration using a BDD-based analysis is currently the most performant approach for multi-threaded

programs in the framework. The heuristics for exploring the abstract state space are tuned to match witnesses from a broad range of verifiers.

The community-based SV-Benchmarks repository is a largest and most divers collection of verification tasks for the language C. We used all verification tasks that were also used by the most recent competition: category *ConcurrencySafety*.

*Internal Validity.* The validator might contain programming bugs, but we based our validator on the infrastructure that is used by the verifier CPA-Seq, which performed extremely well in the recent competitions. Thus, we believe that the implementation has a high quality. Also, previous versions of our validator participated in the competition since SV-COMP 2018. Limitations of the validator were discussed in depth in Sect. 4.2.

The execution of the verification and validation runs was done with BenchExec [22], the (only available) state-of-the-art benchmarking tool, which is also used by the StarExec competition infrastructure and competitions like SV-COMP and Test-Comp. BenchExec is used to enforce the limits and collect measurements for the consumed resources (CPU time and memory).

## 6 Conclusion

While validation of verification results for sequential programs has been thoroughly described in 2015, validation support for multi-threaded programs was not yet described in the literature. This paper closes this gap by describing the (only available) validator for multi-threaded programs, which was already used three times as validator in the competition on software verification (SV-COMP 2018-2020).

In our evaluation, we report the available features that the witnesses produced by several verifiers expose to the validator, and we report the performance. The results are promising, but it would be better for the verification community to have more such validators available: There are six validators for violation witnesses for sequential programs, but only one for multi-threaded programs.

**Data Availability Statement.** We make the violation witnesses, a ready-to-run archive of CPAchecker, and all experimental results (including raw data, tables, and plots) available on a supplementary web site[12] and in a Zenodo archive [14]. The verifiers that participated in SV-COMP 2020 have publicly available archives in a GitLab repository.[11] More witnesses and results from SV-COMP can be found in the archives mentioned in the report [7] (Table 4).

## References

1. Andrianov, P., Mutilin, V., Khoroshilov, A.: Predicate abstraction based configurable method for data race detection in Linux kernel. In: Proc. TMPA, CCIS, vol. 779. Springer (2018). https://doi.org/10.1007/978-3-319-71734-0_2

[12] https://www.sosy-lab.org/research/witnesses-concurrency

2. Artho, C., Havelund, K., Honiden, S.: Visualization of concurrent program executions. In: Proc. COMPSAC, pp. 541–546. IEEE (2007). https://doi.org/10.1109/COMPSAC.2007.236
3. Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkai, P., Štill, V.: Model checking of C and C++ with Divine 4. In: Proc. ATVA, LNCS, vol. 10482, pp. 201–207. Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_14
4. Beyer, D.: Software verification and verifiable witnesses (Report on SV-COMP 2015). In: Proc. TACAS, LNCS, vol. 9035, pp. 401–416. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_31
5. Beyer, D.: Reliable and reproducible competition results with BenchExec and witnesses (Report on SV-COMP 2016). In: Proc. TACAS, LNCS, vol. 9636, pp. 887–904. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_55
6. Beyer, D.: Software verification with validation of results (Report on SV-COMP 2017). In: Proc. TACAS, LNCS, vol. 10206, pp. 331–349. Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_20
7. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Proc. TACAS (2), LNCS, vol. 12079, pp. 347–367. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_21
8. Beyer, D.: SV-Benchmarks: Benchmark set of 9th Intl. Competition on Software Verification (SV-COMP 2020). Zenodo (2020). https://doi.org/10.5281/zenodo.3633334
9. Beyer, D.: Verification witnesses from SV-COMP 2020 verification tools. Zenodo (2020). https://doi.org/10.5281/zenodo.3630188
10. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE, pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351
11. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE, pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
12. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP, LNCS, vol. 10889, pp. 3–23. Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1
13. Beyer, D., Friedberger, K.: A light-weight approach for verifying multi-threaded programs with CPAchecker. In: Proc. MEMICS, EPTCS, vol. 233, pp. 61–71 (2016). https://doi.org/10.4204/EPTCS.233.6
14. Beyer, D., Friedberger, K.: Replication package for article 'Violation witnesses and result validation for multi-threaded programs'. Zenodo (2020). https://doi.org/10.5281/zenodo.3885694
15. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
16. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV, LNCS, vol. 4590, pp. 504–518. Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_51
17. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proc. ASE, pp. 29–38. IEEE (2008). https://doi.org/10.1109/ASE.2008.13

18. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV, LNCS, vol. 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
19. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD, pp. 189–197. FMCAD (2010)
20. Beyer, D., Lemberger, T.: CPA-SymExec: Efficient symbolic execution in CPAchecker. In: Proc. ASE, pp. 900–903. ACM (2018). https://doi.org/10.1145/3238147.3240478
21. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE, LNCS, vol. 7793, pp. 146–162. Springer (2013). https://doi.org/10.1007/978-3-642-37057-1_11
22. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2017). https://doi.org/10.1007/s10009-017-0469-y
23. Beyer, D., Stahlbauer, A.: BDD-based software verification: Applications to event-condition-action systems. Int. J. Softw. Tools Technol. Transfer **16**(5), 507–518 (2014). https://doi.org/10.1007/s10009-014-0334-1
24. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. arXiv/CoRR 1905(08505) (May 2019). https://arxiv.org/abs/1905.08505
25. Beyer, D., Wendler, P.: Reuse of verification results: Conditional model checking, precision reuse, and verification witnesses. In: Proc. SPIN, LNCS, vol. 7976, pp. 1–17. Springer (2013). https://doi.org/10.1007/978-3-642-39176-7_1
26. Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., Marshall, M.S.: GraphML progress report. In: Graph Drawing, LNCS, vol. 2265, pp. 501–512. Springer (2001). https://doi.org/10.1007/3-540-45848-4_59
27. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically generating inputs of death. In: Proc. CCS, pp. 322–335. ACM (2006). https://doi.org/10.1145/1180405.1180445
28. Castaño, R., Braberman, V.A., Garbervetsky, D., Uchitel, S.: Model checker execution reports. In: Proc. ASE, pp. 200–205. IEEE (2017). https://doi.org/10.1109/ASE.2017.8115633
29. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS, LNCS, vol. 2988, pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
30. Csallner, C., Smaragdakis, Y.: Check 'n' crash: Combining static checking and testing. In: Proc. ICSE, pp. 422–431. ACM (2005). https://doi.org/10.1145/1062455.1062533
31. Czech, M., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Predicting rankings of software verification tools. In: Proc. SWAN, pp. 23–26. ACM (2017). https://doi.org/10.1145/3121257.3121262
32. Gadelha, M.Y.R., Ismail, H.I., Cordeiro, L.C.: Handling loops in bounded model checking of C programs via *k*-induction. Int. J. Softw. Tools Technol. Transfer **19**(1), 97–114 (2017). https://doi.org/10.1007/s10009-015-0407-9
33. Gavrilenko, N., Ponce de León, H., Furbach, F., Heljanko, K., Meyer, R.: BMC for weak memory models: Relation analysis for compact SMT encodings. In: Proc. CAV, LNCS, vol. 11561, pp. 355–365. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_19
34. Gennari, J., Gurfinkel, A., Kahsai, T., Navas, J.A., Schwartz, E.J.: Executable counterexamples in software model checking. In: Proc. VSTTE, LNCS, vol. 11294, pp. 17–37. Springer (2018). https://doi.org/10.1007/978-3-030-03592-1_2

35. Greitschus, M., Dietsch, D., Podelski, A.: Loop invariants from counterexamples. In: Proc. SAS, LNCS, vol. 10422, pp. 128–147. Springer (2017). https://doi.org/10.1007/978-3-319-66706-5_7
36. Gunter, E.L., Peled, D.A.: Path exploration tool. In: Proc. TACAS, LNCS, vol. 1579, pp. 405–419. Springer (1999). https://doi.org/10.1007/3-540-49059-0_28
37. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV, LNCS, vol. 8044, pp. 36–52. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
38. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Lazy-CSeq: A lazy sequentialization tool for C (competition contribution). In: Proc. TACAS, LNCS, vol. 8413, pp. 398–401. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_29
39. Inverso, O., Trubiani, C.: Parallel and distributed bounded model checking of multi-threaded programs. In: Proc. PPoPP. ACM (2020)
40. Yin, L., Dong, W., Liu, W., Wang, J.: On scheduling constraint abstraction for multi-threaded program verification. IEEE Trans. Softw. Eng. (2018). https://doi.org/10.1109/TSE.2018.2864122




# JavaSMT 3: Interacting with SMT Solvers in Java

Daniel Baier, Dirk Beyer, and Karlheinz Friedberger

LMU Munich, Munich, Germany


**Abstract.** Satisfiability Modulo Theories (SMT) is an enabling technology with many applications, especially in computer-aided verification. Due to advances in research and strong demand for solvers, there are many SMT solvers available. Since different implementations have different strengths, it is often desirable to be able to substitute one solver by another. Unfortunately, the solvers have vastly different APIs and it is not easy to switch to a different solver (lock-in effect). To tackle this problem, we developed JavaSMT, which is a solver-independent framework that unifies the API for using a set of SMT solvers. This paper describes version 3 of JavaSMT, which now supports eight SMT solvers and offers a simpler build and update process. Our feature comparisons and experiments show that different SMT solvers significantly differ in terms of feature support and performance characteristics. A unifying Java API for SMT solvers is important to make the SMT technology accessible for software developers. Similar APIs exist for other programming languages.

**Keywords:** Satisfiability Modulo Theories · SMT Solver · Java · API

## 1 Introduction

SMT solvers [6, 21] are used in a multitude of applications, e.g., in formal software analysis, where automated test-case generation [7, 16, 29, 30], SMT-based algorithms for software verification [10, 34], and interactive theorem proving [27, 44] are used. Applications and users rely on efficiency and expressiveness (supported SMT theories) to compute reasonable results in time. For application developers, the usability and API of the solver are also important aspects, and some features needed in applications, such as interpolation or optimization, are not available in some solvers.

Using the solver's own API directly makes it difficult to switch to another solver without rewriting extensive parts of the application, as there is no standardized binary API for SMT solvers. The SMT-LIB2 standard [4] improves this issue by defining a common language to interact with SMT solvers. However, this communication channel does not define a solver interface for special features like optimization or interpolation.[1] Additionally, the application has to parse the data provided by the SMT solver on its own, and this of course slightly changes from solver to solver.

[1] A proposal for adding interpolation queries exists since 2012, see https://ultimate.informatik.uni-freiburg.de/smtinterpol/proposal.pdf .


A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 195–208, 2021.
https://doi.org/10.1007/978-3-030-81688-9_9

JavaSMT [37] provides a common API layer across multiple back-end solvers to address these problems. Our Java-based approach creates only minimal overhead, while giving access to most solver features. JavaSMT is available under the Apache 2.0 License on GitHub.[2]

**Contribution.** Our contribution consists of three parts:

- We integrated more SMT solvers into the API framework JavaSMT (new: Boolector [43], CVC4 [5], and Yices2 [25]).
- We simplified the steps to get started using JavaSMT, by including support for more operating systems (new: MacOS and Windows) and more build techniques (new: Ant and Maven).
- We evaluated the performance of several algorithms for software verification to show that different SMT solvers have different strengths.

**Outline.** This paper first provides a brief overview of JavaSMT in Sect. 2, explaining the inner structure and features. Sect. 3 discusses the development since the previous publication [37]: more integrated SMT solvers and extended support for operating systems and build processes. Sect. 4 describes a case study, based on SMT-based algorithms [10] in a common verification framework.

**Related Work.** SMT-LIB2 [4] is the established standard format for exchanging SMT queries. It provides simple usage, is easy to debug, and widely known in the community. However, it requires extra effort to parse and transform formulas in the user application. Features like optimization, interpolation, and receiving nested parts of formulas are not defined by the standard, such that some SMT solvers provide their own individual solution for that. Alternatively, several SMT solvers already come with their special bindings for some programming languages. Most SMT solvers are written in C/C++, so interacting with them in these low-level languages is the easiest way. However, the support for higher-level languages is sparse. The most prominent language binding for several SMT solvers is Python, as it directly allows the access to C code and avoids automated memory management operations like asynchronous garbage collection. Bindings for Java are available for some SMT solvers, such as MathSAT5 and Z3, but missing, unsupported, or unmaintained for others, such as Boolector and CVC4.

In the following, we discuss libraries, similar to JavaSMT, that provide access to several underlying SMT solvers via a common user interface in different popular languages, and their binding mechanism, i.e., whether the solver interaction is based on a native interface or text-based on SMT-LIB2. With SMT-LIB2, an arbitrary SMT solver can be queried, but the interaction happens through communicating processes and the solver is mostly limited to features defined in the standard. Accessing a native interface directly allows to support more features of the underlying solver, e.g., using callbacks, simplifying formulas, or eliminating quantifiers.

Table 1 provides an overview of the libraries for interacting with SMT solvers. We enumerate several special features that are not available in some libraries,

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

Table 1: Comparison of different interface libraries for SMT solvers

| | Reference | Language | Native API | SMT-LIB2 | Unsat Cores | Interpolation | Optimization | Formula Decomposition | Project - Forks | Project - Stars | Project - Year Latest Commit |
|---|---|---|---|---|---|---|---|---|---|---|---|
| JavaSMT | [37] | Java | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | 22 | 90 | 2021 |
| PySMT | [28] | Python | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | 99 | 363 | 2021 |
| SMT Kit | | C/C++ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | 4 | 36 | 2014 |
| Smt-Switch | [38] | C/C++ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | 15 | 40 | 2021 |
| jSMTLIB | [20] | Java | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | 15 | 21 | 2020 |
| metaSMT | [45] | C/C++ | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | 19 | 43 | 2016 |
| rsmt2 | | Rust | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | 10 | 24 | 2021 |
| SBV | | Haskell | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | 17 | 134 | 2021 |
| Scala SMT-LIB | | Scala | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | 18 | 44 | 2021 |
| ScalaSMT | [17] | Scala | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | 1 | 4 | 2019 |
| what4 | | Haskell | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | 5 | 97 | 2021 |

such as unsat cores, interpolation, or optimization queries. Those features depend on the support by the underlying SMT solver, but can be provided in general by an API on top of them. Most libraries use their own formula representation and not just wrap the objects provided by the SMT solver. This potentially allows for easier formula decomposition and inspection, e.g., by using the visitor pattern. JavaSMT directly provides formula decomposition if available in the SMT solver. The provided numbers of forks and stars of the project repositories on GitHub or Bitbucket can be seen as a measurement of popularity.

PySMT [28] is a Python-based project and aims at rapid prototyping of algorithms using the native API of the installed SMT solvers. It has the ability to perform formula manipulation without a back-end SMT solver and additionally supports the conversion of boolean formulas to plain SAT problems and then apply a SAT solver or a BDD library. This approach comes with the drawback of a noticeable memory overhead and performance of an interpreted language. metaSMT [45], SMT Kit, and Smt-Switch [38] provide solver-agnostic APIs for interacting with various SMT solvers in C/C++ to focus on the application instead of the solver integration. jSMTLIB [20], Scala SMT-LIB, and ScalaSMT [17] are solver-independent libraries written in Java or Scala and interact via SMT-LIB2 with SMT solvers. Scala SMT-LIB and ScalaSMT allow to use an additional domain-specific language to interact with SMT solvers and rewrite Scala syntax into valid SMT-LIB2 and back. Both partially extend the SMT-LIB2 standard, e.g., by offering the ability to overload operators or receive interpolants. SBV and what4 are generic Haskell libraries based on process interaction via SMT-LIB2 and support several SAT and SMT solvers. rsmt2 offers a generic Rust library that currently supports three SMT solvers.

## 2 JavaSMT's Architecture and Solver Integration

In the following, we describe the architecture of JavaSMT and its main concepts. Afterwards, we give an overview of the integrated SMT solvers and their features. The architecture did not significantly change, but we added a few new SMT solvers, as shown in Fig. 1.

**Architecture.** JavaSMT provides a common API for various SMT solvers. The architecture, shown in Fig. 1, consists of several components: As common context, we use a `SolverContext` that loads the underlying SMT solver and defines the scope and lifetime of all created objects. As long as the context is available, we track memory regions of native SMT-solver libraries. When the context is closed, the corresponding memory is freed and garbage collection wipes all unused objects. Within a given context, JavaSMT provides `FormulaManager`s for creating formulas in various theories and `ProverEnvironment`s for solving SMT queries.

A `FormulaManager` allows to create symbols and formulas in the corresponding theories and provides a type-safe way to combine symbols and formulas in order to encode a more complex SMT query. We support the structural analysis (like splitting a formula into its components or counting all function applications in a formula) and transformations (like substituting symbols or applying equisatisfiable simplifications) of formulas.

Each `ProverEnvironment` represents a solver stack and allows to push/pop boolean formulas and check them for satisfiability (the hard part). This follows the idea of incremental solving (if the underlying SMT solver supports it). After a satisfiability check, the `ProverEnvironment` provides methods to receive a model, interpolants, or an unsatisfiable core for the given formula.

JavaSMT guarantees that formulas built with a single `FormulaManager` can be used in several `ProverEnvironment`s, e.g., the same formula can be pushed onto and solved within several distinct `ProverEnvironment`s. The interaction with independent `ProverEnvironment`s works from multiple threads. However, some SMT solvers require synchronization (e.g., locking for an interleaved usage) and other solvers do not require external synchronization (this allows concurrent usage).

**SMT-Solver Integration and Bindings.** Of the eight SMT solvers that are available in JavaSMT, only Princess [46] and SMTInterpol [18] were 'easy' to integrate, as they are written in Scala and Java, respectively. Those solvers also use the available memory management and garbage collection of the Java Virtual Machine (JVM). All other solvers are written in C/C++ and need a Java Native Interface (JNI) wrapper to interface with JavaSMT. Z3 [40] and CVC4 [5] provide their own Java wrappers, while the bindings used for MathSAT5 [19], Boolector [42], and Yices2 [25] are maintained by us. Those bindings are self-written or partially based on a version of the solver developers, extended with exception handling, and usable for debugging in JavaSMT. By providing language bindings for solvers in our library, we relieve the solver developers from this burden, and the implementation of exception handling and memory management is done in an efficient and common manner across several solvers.

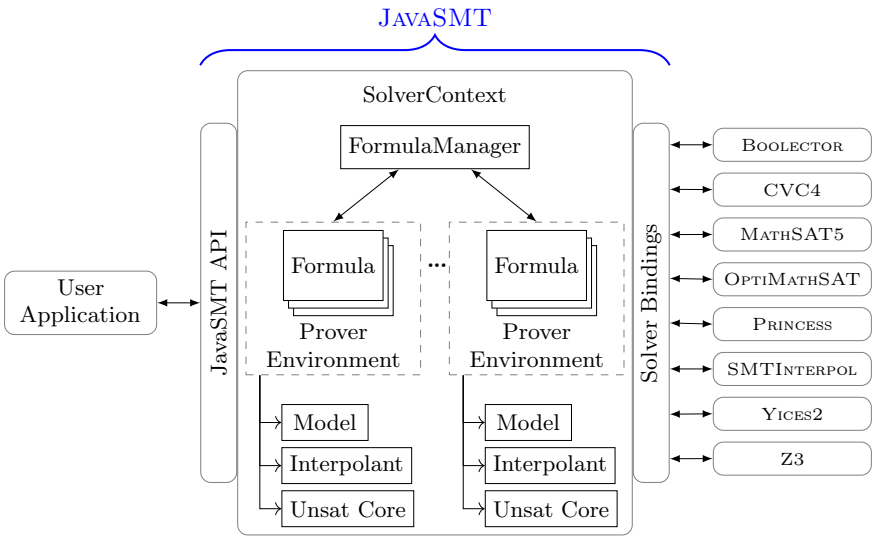



Fig. 1: Overview of JavaSMT

Table 2: Size (LOC) of the Java-based solver wrappers and native solver bindings

| | Boolector | CVC4 | MathSAT5 | OptiMathSAT | Princess | SMTInterpol | Yices2 | Z3 |
|---|---|---|---|---|---|---|---|---|
| Java-based Wrapper | 1644 | 1918 | 3229 | 3229 | 2042 | 2117 | 2728 | 2674 |
| JNI Bindings | 3136 | | 1388 | 1508 | | | 1598 | |

Table 2 lists the size (lines of code) of the wrappers to integrate each solver in JavaSMT, in order to get a rough impression of the required effort to get a solver and its bindings usable in JavaSMT. The size information consists of two parts, namely the JNI bindings that are written in C/C++ and the Java code that implements the necessary interfaces of JavaSMT. An expressive solver API (like MathSAT5 or OptiMathSAT [47]) needs more code for their binding, with only a small increment in complexity compared to other solver bindings.

Note that the evolution of JavaSMT depends on the evolution of the underlying SMT solvers. Z3 is well-known, has a large user group, and an active development team. Yet, interpolation support for Z3 was dropped with release 4.8.1.[3] Bitwuzla [41] is the successor of the SMT solver Boolector, for which the developers still provide small fixes. Bitwuzla can be supported in JavaSMT in the future. CVC4 has been developed further to CVC5. However, the maintainers

[3] https://github.com/Z3Prover/z3/releases/tag/z3-4.8.1

dropped the existing Java API, partially because of issues with the Java garbage collection, and plan to replace it.[4] Yices2 is also actively maintained and adds new features regularly. For example, the developers added support for third-party SAT solvers such as CaDiCaL and CryptoMiniSat [48].

## 3 New Contributions in JavaSMT 3

This section describes the improvements over the JavaSMT version from five years ago [37], split into two parts. First, we describe newly integrated solvers and theory features. Second, we provide information about the build process.

**Support for Additional SMT Solvers.** JavaSMT 3 provides access to eight SMT solvers. Besides the solvers that were already integrated before, MathSAT5, OptiMathSAT, Z3, Princess, and SMTInterpol, the user can now additionaly use Boolector, CVC4, and Yices2. Table 3 lists available theories and important features supported by each individual solver. Boolector is specialized in Bitvector-based theories, but does not support the Integer theory. It is shipped with several back-end SAT solvers, from which the user can choose a favorite: CaDiCaL, CryptoMiniSat [48], Lingeling, MiniSat [26], and PicoSAT [13]. All solvers support the input of plain SMT-LIB2 formulas. However, the feature most requested by JavaSMT users is the input and output of SMT queries via the API, i.e., parsing and printing boolean formulas for a given context. This feature is required for (de-)serializing formulas to disk, for network transfer, and to translate formulas from one solver to another one. This feature is unfortunately missing for the newly integrated solvers, even though each solver internally already contains code for parsing and printing SMT-LIB2 formulas.

For formula manipulation, JavaSMT accesses the components of a formula, e.g., operators and operands. We do not require full access to the internal data structures of the SMT solvers, but only limited access to the most basic parts. Only Boolector does not provide the necessary API.

**Build Simplification.** JavaSMT 3 also supports more operating systems than before. Besides the existing support for Linux, we started to provide pre-compiled binaries for MacOS and Windows for more than half of the available solvers. This simplifies the initial steps for new users, which previously were required to compile and link the solvers on their own. This was an involving task, because of the diversity of build systems and dependencies of each solver.

In addition to this, we now offer direct support for two popular build systems for Java applications, namely Ant and Maven. JavaSMT comes with several examples and documentation, such that the mentioned build systems can be used to set up JavaSMT in a ready-to-go state on most systems. This eliminates the need for complex manual set up of dependencies and eases the use of JavaSMT and the SMT solvers.

[4] https://github.com/cvc5/cvc5/issues/5018

Table 3: SMT theories and features supported by SMT solvers in JavaSMT 3

| | | Boolector | CVC4 | MathSAT5 | OptiMathSAT | Princess | SMTInterpol | Yices2 | Z3 |
|---|---|---|---|---|---|---|---|---|---|
| SMT Theories | Integer | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Rational | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Array | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ |
| | Bitvector | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ |
| | Float | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ |
| | UF | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Quantifier | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ |
| Features | Incremental Solving | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Model | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Assumption Solving | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ |
| | Interpolation | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ |
| | Optimization | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ |
| | UnsatCore | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | UnsatCore with Assumptions | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ |
| | SMT-LIB2 (plain text input) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | SMT-LIB2 (via API) | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ |
| | Quantifier Elimination | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ |
| | Formula Decomposition | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

## 4 Evaluation

Frameworks that provide a unified API to SMT solvers (such as JavaSMT, PySMT, and ScalaSMT) are necessary because the characteristics of the SMT solvers vary a lot. In the evaluation we provide support for this argument.

We inlined a discussion of the features already in the previous section. Table 3 provides the overview of supported theories and shows that certain theories are available only for a subset of SMT solvers. The table also shows that there are several features that restrict the choice of SMT solvers for certain applications.

In terms of performance, we evaluate JavaSMT 3 as a component of CPAchecker [11], which is an open-source software-verification framework [5] that provides a range of different SMT-based algorithms for program analysis [10] and encoding techniques for program control flow [8, 12]. We compare three well-known and successful SMT-based algorithms for software model checking and show that — when using the same algorithm and identical problem encoding — the performance result of an analysis depends on the used SMT solver. Some

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

algorithms depend on special features of the SMT solver, e.g., to provide a certain type of formula (such as interpolants) and operation on a formula (such as access to subformulas). There are SMT solvers that can not be used for some algorithms.

We aim to show that depending on the feature set of the SMT solvers, it is important to support a common API, and additionally, that using the text-based interaction via SMT-LIB2 is not an efficient solution, when it comes to formula analysis like adding additional information into a formula.

**Benchmark Programs.** We evaluate the usage of JavaSMT on a large subset of the SV-benchmark suite [6] containing over 1 000 verification tasks. To have a broad variation of benchmark tasks, we include reachability problems from the categories *BitVectors*, *ControlFlow*, *Heap*, and *Loops*.

*BitVectors* depends on bit-precise reasoning and thus, the SMT solver needs to support Bitvector logic. *Heap* depends on modeling heap memory access, e.g., which is either encoded in the theory of Arrays or as Uninterpreted Functions. The category *Loops* contains tasks where the state space is potentially quite large.

**Experimental Setup.** We run all our experiments on computers with Intel Xeon E3-1230 v5 CPUs with 3.40 GHz, and limit the CPU time to 15 min and the memory to 15 GB. We use CPAchecker revision r36714, which internally uses JavaSMT 3.7.0-73. The time needed for transforming the input program into SMT queries is rather small compared to the analysis time. Additionally, the progress of an algorithm depends on the result (e.g., model values or interpolants) returned from an SMT solver, thus we do not explicitly extract the run time required by the SMT solver itself for answering the satisfiability problem, but we measure the complete CPU time of CPAchecker for the verification run.

**Analysis Configuration.** We use three different SMT-based algorithms for software verification [10]. The first approach is bounded model checking (BMC) [14, 15], which is applied in software and hardware model checking since many years. In this approach, a verification problem is encoded as single large SMT query and given to the SMT solver. No further interaction with the SMT solver is required. In our evaluation, we use a loop bound $k = 10$, which limits the size of the SMT query.

The second approach is $k$-induction [9, 24], which extends BMC, and which uses auxiliary invariants to strengthen the induction hypothesis. In this approach, the algorithm generates several SMT queries (base case, inductive-step case, each with increasing loop bound) and uses an invariant generator that provides the auxiliary invariants. We use an interval-based invariant generator that provides not only the invariants, but also information about pointers and aliases, which must be inserted into the SMT formula using the formula visitor.

The third approach is predicate abstraction [3, 12, 31, 35], which uses Craig interpolation [22, 32, 39] to compute predicate abstractions of the program. This approach does not only query the SMT solver multiple times, but also uses (sequential) interpolation, which is currently supported only by MathSAT5, Princess, and SMTInterpol.

[6] https://github.com/sosy-lab/sv-benchmarks

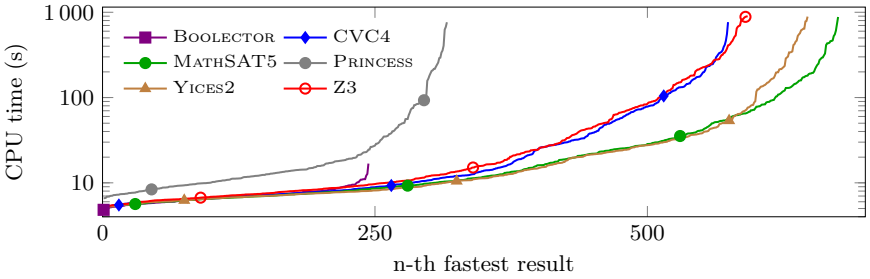



Fig. 2: Quantile plot for the runtime of $k$-induction with several SMT solvers

All approaches are executed in two configurations, depending on the used encoding of program statements: First, we apply a bitvector-based encoding that precisely models bit-precise arithmetics and overflows of the program. Second, an encoding based on linear integer arithmetic is used, which approximates the concrete program execution and is sufficient for some programs.

**Solver Configuration.** Overall, we aim to show that each solver provides a unique fingerprint of features and results. We aim for a precise program analysis and thus configure the SMT solvers to be as precise as possible, but with a reasonable configuration for each solver (i.e., without using a feature combination that is unsupported by the SMT solver).

SMTInterpol does not support efficient solving of SMT queries in Bitvector logic, thus, it is configured to use only Integer logic. Boolector misses Integer logic, thus, it is applied only to the bit-precise configurations. Additionally, this SMT solver does not support formula inspection and decomposition, which is required by several components in $k$-induction, e.g., to encode proper pointer aliasing for the program analysis. While the code for formula inspection is called quite often, its influence on the results for the selected benchmark tasks is small. In order to be comparable as far as possible, we deactivate pointer aliasing when using Boolector. Yices2 misses proper support for Array logic, thus, we use a UF-based encoding of heap memory as alternative for this solver, which results in a slightly unsound analysis, but a comparable formula size and run time.

**Results and Discussion.** Figure 2 provides the quantile plot for the results of $k$-induction configurations with bit-precise encoding using several SMT solvers. The plot shows the CPU time for valid analysis results, i.e., proofs or counterexamples found, for both expected results true and false. We aim for providing all result that are useful for a user and do not show results where the tool (or SMT solver) crashes or runs out of resources. We do not subtract the run time required for the framework CPAchecker itself (which starts a Java virtual machine), as we assume it to be comparable per program task; we are only interested in the asymptotics in this evaluation. The overall performance of SMT solvers is similar for simple verification tasks, i.e., those with a small run time in the analysis. For difficult tasks with harder SMT queries, the differences of the SMT solvers emerge. When applying $k$-induction, the analysis inserts additional constraints into the

Table 4: Run time for using different SMT solvers for bounded model checking ('BMC'), k-induction ('KI'), and predicate abstraction ('PA') with the theories of Bitvectors ('BV') and Integers ('Int'); CPU time given in seconds with two significant digits, ' TO' indicates timeouts (900 s), ' ERR' indicates errors, and empty cells indicate that the theory or interpolation was not supported

| Verification Task | s3_srvr.blast.07.i.cil-2 | byte_add_1-1 | ps6-ll_valuebound100 | s3 | diamond_1-1 | modulus-2 | jain_5-2 | s3_clnt_1.cil-2 | diskperf_simpl1.cil | rule57_ebda_blast |
|---|---|---|---|---|---|---|---|---|---|---|
| Algorithm | BMC | BMC | KI | KI | KI | KI | PA | PA | PA | PA |
| Encoding | Int | BV | Int | Int | BV | BV | Int | Int | BV | BV |
| Boolector | | **5.8** | | | ERR | ERR | | | | |
| CVC4 | 340 | 6.4 | TO | TO | 110 | TO | | | | |
| MathSAT5 | 17 | 7.8 | 200 | 53 | 60 | 54 | TO | **11** | **16** | **7.1** |
| Princess | TO | TO | 530 | TO | 260 | TO | **38** | 160 | TO | ERR |
| SMTInterpol | 50 | | TO | 140 | | | TO | 13 | | |
| Yices2 | **14** | 7.7 | 340 | **23** | **34** | 28 | | | | |
| Z3 | 15 | 6.7 | **130** | 66 | 43 | **21** | | | | |

SMT formula and requires the SMT solver to allow access to components of existing formulas. As Boolector misses this specific feature, $k$-induction cannot be very effective here. Other SMT solvers are the preferred choice.

Table 4 contains some example tasks from all used algorithms and encodings, where the difference between distinct SMT solvers is noteworthy. Choosing the optimal SMT solvers for an arbitrary problem task is not obvious.

## 5 Conclusion

We contribute JavaSMT 3, the third generation of the unifying Java API for SMT solvers. The package now contains more SMT solvers, an improved build process, and support for MacOS and Windows. The project has over 20 contributors, 2 500 commits, and overall about 41 000 lines of code.[7] JavaSMT is used in Java applications (e.g., [23, 33, 36]) as a solution to combine convenience and performance for the interaction with SMT solvers, or to switch between different solvers and compare them [11, 49]. The most prominent application using JavaSMT is the verification framework CPAchecker (a widely-used software

[7] https://www.openhub.net/p/java-smt

project [8] with 73 forks on GitHub alone), for which JavaSMT was originally developed. In the future, we plan to support more SMT solvers, operating systems, and hardware architectures, while keeping the user interface stable. We hope that even more researchers and developers of Java applications can benefit from SMT solving via a convenient and powerful API.

**Data Availability Statement.** All benchmark tasks for evaluation, configuration files, a ready-to-run version of our implementation, and tables with detailed results are available in our reproduction package on Zenodo as virtual machine [1] and as ZIP archive [2]. The source code of the open-source library JavaSMT [37] is available in the project repository; see `https://github.com/sosy-lab/java-smt`.

**Funding.** This project was supported by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY).

## References

1. Baier, D., Beyer, D., Friedberger, K.: Reproduction package (VM) for article 'JavaSMT 3: Interacting with SMT solvers in Java'. Zenodo (2021). https://doi.org/10.5281/zenodo.4708050
2. Baier, D., Beyer, D., Friedberger, K.: Reproduction package (ZIP) for article 'JavaSMT 3: Interacting with SMT solvers in Java'. Zenodo (2021). https://doi.org/10.5281/zenodo.4865175
3. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking C programs. In: Proc. TACAS. pp. 268–283. LNCS 2031, Springer (2001). https://doi.org/10.1007/3-540-45319-9_19
4. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Proc. SMT (2010)
5. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proc. CAV. pp. 171–177. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14
6. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Model Checking, pp. 305–343. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_11
7. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. ICSE. pp. 326–335. IEEE (2004). https://doi.org/10.1109/ICSE.2004.1317455
8. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proc. FMCAD. pp. 25–32. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351147
9. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
10. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6

[8] `https://github.com/sosy-lab/cpachecker`

11. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
12. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD. pp. 189–197. FMCAD (2010)
13. Biere, A.: PicoSAT Essentials. JSAT **4**(2-4), 75–97 (2008). https://doi.org/10.3233/SAT190039
14. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
15. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58**, 117–148 (2003). https://doi.org/10.1016/S0065-2458(03)58003-2
16. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008)
17. Cassez, F., Sloane, A.M.: ScalaSMT: Satisfiability modulo theory in Scala (tool paper). In: Proc. SCALA. pp. 51–55. ACM (2017). https://doi.org/10.1145/3136000.3136004
18. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An interpolating SMT solver. In: Proc. SPIN. pp. 248–254. LNCS 7385, Springer (2012). https://doi.org/10.1007/978-3-642-31759-0_19
19. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Proc. TACAS. pp. 93–107. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
20. Cok, D.R.: jSMTLIB: Tutorial, validation, and adapter tools for SMT-LIBv2. In: Proc. NFM. pp. 480–486. LNCS 6617, Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_36
21. Cok, D.R., Déharbe, D., Weber, T.: The 2014 SMT competition. JSAT **9**, 207–242 (2016)
22. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957). https://doi.org/10.2307/2963593
23. Demarchi, S., Menapace, M., Tacchella, A.: Automating elevator design with satisfiability modulo theories. In: Proc. ICTAI. pp. 26–33. IEEE (2019). https://doi.org/10.1109/ICTAI.2019.00013
24. Donaldson, A.F., Haller, L., Kröning, D., Rümmer, P.: Software verification using k-induction. In: Proc. SAS. pp. 351–368. LNCS 6887, Springer (2011). https://doi.org/10.1007/978-3-642-23702-7_26
25. Dutertre, B.: Yices 2.2. In: Proc. CAV. pp. 737–744. LNCS 8559, Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49
26. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. SAT. pp. 502–518. LNCS 2919, Springer (2003). https://doi.org/10.1007/978-3-540-24605-3_37
27. Ernst, G., Huisman, M., Mostowski, W., Ulbrich, M.: VerifyThis: Verification competition with a human factor. In: Proc. TACAS. pp. 176–195. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_12
28. Gario, M., Micheli, A.: PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In: Proc. SMT (2015)
29. Godefroid, P., Sen, K.: Combining model checking and testing. In: Handbook of Model Checking, pp. 613–649. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_19

30. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proc. NDSS. The Internet Society (2008)
31. Graf, S., Saïdi, H.: Construction of abstract state graphs with Pvs. In: Proc. CAV. pp. 72–83. LNCS 1254, Springer (1997). https://doi.org/10.1007/3-540-63166-6_10
32. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL. pp. 232–244. ACM (2004). https://doi.org/10.1145/964001.964021
33. Ibrhim, H., Khattab, S., Elsayed, K., Badr, A., Nabil, E.: A formal methods-based rule verification framework for end-user programming in campus building automation systems. Building and Environment **181**, 106983 (2020). https://doi.org/10.1016/j.buildenv.2020.106983
34. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41**(4) (2009). https://doi.org/10.1145/1592434.1592438
35. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Handbook of Model Checking, pp. 447–491. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_15
36. Joshaghani, R., Black, S., Sherman, E., Mehrpouyan, H.: Formal specification and verification of user-centric privacy policies for ubiquitous systems. In: Proc. IDEAS. pp. 31:1–31:10. ACM (2019). https://doi.org/10.1145/3331076.3331105
37. Karpenkov, E.G., Friedberger, K., Beyer, D.: JavaSMT: A unified interface for SMT solvers in Java. In: Proc. VSTTE. pp. 139–148. LNCS 9971, Springer (2016). https://doi.org/10.1007/978-3-319-48869-1_11
38. Mann, M., Wilson, A., Tinelli, C., Barrett, C.W.: SMT-Switch: A solver-agnostic C++ API for SMT solving. arXiv/CoRR (2007.01374) (2020), https://arxiv.org/abs/2007.01374
39. McMillan, K.L.: Interpolation and model checking. In: Handbook of Model Checking, pp. 421–446. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_14
40. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS. pp. 337–340. LNCS 4963, Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
41. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. arXiv/CoRR (2006.01621) (2020), https://arxiv.org/abs/2006.01621
42. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. J. Satisf. Boolean Model. Comput. **9**(1), 53–58 (2014). https://doi.org/10.3233/sat190101
43. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC, and Boolector 3.0. In: Proc. CAV. pp. 587–595. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_32
44. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS 2283, Springer (2002). https://doi.org/10.1007/3-540-45949-9
45. Riener, H., Haedicke, F., Frehse, S., Soeken, M., Große, D., Drechsler, R., Fey, G.: metaSMT: Focus on your application and not on solver integration. Int. J. Softw. Tools Technol. Transf. **19**(5), 605–621 (2017). https://doi.org/10.1007/s10009-016-0426-1
46. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Proc. LPAR. pp. 274–289. LNCS 5330, Springer (2008). https://doi.org/10.1007/978-3-540-89439-1_20
47. Sebastiani, R., Trentin, P.: OptiMathSAT: A tool for optimization modulo theories. In: Proc. CAV. pp. 447–454. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_27
48. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) Proc. SAT. pp. 244–257. LNCS 5584, Springer (2009). https://doi.org/10.1007/978-3-642-02777-2_24

208 D. Baier, D. Beyer, and K. Friedberger

49. Sprey, J., Sundermann, C., Krieter, S., Nieke, M., Mauro, J., Thüm, T., Schaefer, I.: SMT-based variability analyses in FeatureIDE. In: Proc. VaMoS. pp. 6:1–6:9. ACM (2020). https://doi.org/10.1145/3377024.3377036