

Sudeep Kanav

# Exploring Cooperative Verification

Survey, Tools, and Experiments

Dissertation an der Fakultät für Mathematik, Informatik und Statistik der Ludwig-Maximilians-Universität München • Software and Computational Systems Lab • Eingereicht von Sudeep Kanav am 19. April 2023

Sudeep Kanav

# Exploring Cooperative Verification

## Survey, Tools, and Experiments

1. Gutachter: Prof. Dr. Dirk Beyer
2. Gutachter: Univ.-Prof. Dr. Johannes Kinder
3. Gutachter: Prof. Dr. Philipp Rümmer

Tag der Einreichung: 19. April 2023

Tag der mündlichen Prüfung: 23. April 2024

Dissertation an der Fakultät für Mathematik, Informatik und Statistik der Ludwig-Maximilians-Universität München • Software and Computational Systems Lab



*Eidesstattliche Versicherung*

(Siehe Promotionsordnung vom 12.07.11, § 8, Abs. 2 Pkt. .5.)

Hiermit erkläre ich an Eidesstatt, dass die Dissertation von mir selbstständig und ohne unerlaubte Beihilfe angefertigt ist.

Kanav, Sudeep  
Name, Vorname

München, 19. April 2023  
Ort, Datum

Unterschrift Doktorand/in

## *Zusammenfassung*

Es gibt kein Patentrezept für die Softwareverifikation: In den letzten Jahrzehnten wurden viele erfolgreiche Werkzeuge und Techniken entwickelt, jedoch kommen diese mit ihren eigenen Stärken und Schwächen. Für bestimmte Anwendungsbereiche sind einige Techniken und Werkzeuge daher besser geeignet als andere.

Deshalb liegt es nahe, verschiedene Kombinationen von bestehenden Verifikationstechniken auszuprobieren. Diese Kombinationen können in bestehende Werkzeuge integriert oder direkt aus bereits verfügbaren Komponenten zusammengesetzt werden. Hierbei tauschen die Komponenten unter Umständen Informationen aus, um schneller ein Ergebnis zu erzielen.

Diese Arbeit konzentriert sich auf kooperative Verifikation, also die Kombination von bestehenden Verifikationskomponenten. Dabei arbeiten die Komponenten als Team; sie tauschen Informationen untereinander aus, um gemeinsam das zugrundeliegende Verifikationsproblem zu lösen.

Zu Beginn der Arbeit werden bereits existierende kooperative Techniken in einer systematischen Literaturstudie eingeordnet, verstanden und kategorisiert. Das Ergebnis der Studie legt nahe, dass viele kooperative Techniken ad hoc implementiert werden. Basierend auf diesem Erkenntnis, wurde CoVeriTeam entwickelt. Es erlaubt, dass kooperative Ansätze innerhalb eines Frameworks systematisch konstruiert und ausgeführt werden können. Mithilfe von CoVeriTeam können verschiedene Kombinationen von Techniken auch evaluiert werden. Das Ergebnis: Tatsächlich übertrifft die Kombination der Techniken die Leistungen der einzelnen Komponenten. Zudem vereinfacht CoVeriTeam die Nutzung von bestehenden Komponenten durch einen Webservice, welcher es Nutzern erlaubt, eine Vielzahl von Verifikationswerkzeugen und -techniken auszuprobieren – ganz ohne diese installieren zu müssen.

Die Ergebnisse dieser Forschungsarbeit werden bereits erfolgreich in wichtigen Bereichen der formalen Verifikation eingesetzt: (1) CoVeriTeam wurde in einer Forschungsarbeit für einen komponentenbasierten Ansatz für das sogenannte Counterexample Guided Abstraction Refinement (CEGAR) eingesetzt, (2) Zwei Werkzeuge, die mit CoVeriTeam erstellt wurden, haben im elften internationalen Wettbewerb für Softwareverifikation 2022 (SV-COMP 2022) teilgenommen, (3) Unser CoVeriTeam Service wurde vor der Ausführung der SV-COMP 2023 benutzt, um teilnehmenden Teams vorab unkompliziert mitzuteilen, ob die eingereichte Software auf der vorhandenen Infrastruktur ausgeführt werden kann, (4) Das Huawei Dresden Research Center auf CoVeriTeam aufbauen eine Schnittstelle für Abgestellte bereitgestellt, welche es erlaubt Verifikationswerkzeuge ohne vorherige Installation benutzen können.

und im HDRC wurde eine Schnittstelle entwickelt, welche es erlaubt, VWZ ohne vorherige Installation benutzen zu können und die Fähigkeiten von VWZ zu demonstrieren.

## Abstract

No silver bullet exists for software verification: Many successful tools and techniques have been developed over the past few decades, but none of them is sufficient to tackle the problem on its own. Each of these successful tools and techniques is strong in some specific areas but weak in others.

With this background, it is natural to try out combinations of verification techniques. These combinations could either be integrated or constructed using off-the-shelf components, where the components may exchange information.

This thesis explores the area of *cooperative verification*. A cooperative verification technique is a combination of verification techniques where different *off-the-shelf* components work together, as a team, sharing information with each other with the goal of solving a verification problem.

As a first step, we reviewed scientific literature to understand and systematize the knowledge about cooperative verification. We found that most of these combinations were developed *ad hoc*. To address this gap, we developed CoVeriTeam, a tool for systematic construction and execution of combinations of verification tools. We conducted an extensive evaluation of tool combinations constructed using CoVeriTeam to investigate their performance in comparison to the standalone tools when solving verification problems. Furthermore, to improve the accessibility of verification tools, we have developed a web service for CoVeriTeam that allows users to execute verification tools and their combinations remotely.

Our work is already having an impact: (1) CoVeriTeam was used in research work to construct an off-the-shelf component-based counterexample-guided abstraction-refinement tool, (2) two tools constructed using CoVeriTeam participated in the 11th Competition on Software Verification held in 2022, (3) CoVeriTeam Service was used in integration scripts for the 2023 competitions on software verification and testing to check if the submitted tools can be successfully executed, and (4) a user interface based on an instance of CoVeriTeam Service has been developed at Huawei Dresden Research Center to demonstrate the capabilities of verification tools and allow other teams at Huawei to use these tools without needing to install them.



### *Acknowledgements*

The list of people I have to thank is long, as I sought help from a lot of people. This thesis would have remained unfinished without their support.

This PhD thesis would have been impossible without my supervisor Dirk Beyer. I am grateful to him for giving me the opportunity to pursue doctoral studies and guiding me through this journey. I thank him for sharing his experiences and discussions about broader research world and practices.

During my research, I was lucky to find collaborators who were fun to work with and contributed to successful research projects: Tobias Kleinert, Cedric Richter, and Henrik Wachowitz. I thank Nian-Ze Lee, Thomas Lemberger, Philipp Wendler, and Stefan Winter for the discussions about my research and feedback on the drafts of my papers, helping me improve my work.

I would also like to thank my colleagues at the SoSy Lab for their role in creating and maintaining an environment conducive for research. In alphabetical order: Thomas Bunk, Po-Chun Chien, Gidon Ernst, Matthias Kettl, Nian-Ze Lee, Thomas Lemberger, Martin Spießl, Henrik Wachowitz, Philipp Wendler, and Stefan Winter.

I thank Dhiraj Kumar Gulati and Thomas Bunk for proofreading and providing valuable feedback on my thesis.

Before this success, I have failed in one attempt at PhD. I cannot forget people who supported me at that time, and motivated me to find another opportunity resulting in me joining SoSy Lab. I thank Vincent Aravantinos, Dhiraj Kumar Gulati, Levi Lúcio, Hernan Ponce De Leon, Tarik Terzimehić, and Marco Volpe. Hernan also helped me find an internship during the last phase when my funding was drying up, providing me means to sustain myself.

I also want to thank Johannes Kinder and Philipp Rümmer for their time and effort to review my thesis.

I dedicate this work to my parents Hem Lata Sharma and Raj Kumar Sharma, who have sacrificed a lot to provide me and my siblings many more opportunities than they ever got.





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Contributions . . . . .	2
1.2	Impact . . . . .	2
1.3	Structure . . . . .	3
<b>2</b>	<b>Definitions</b>	<b>5</b>
2.1	Example . . . . .	5
2.2	Actors and Artifacts . . . . .	6
2.3	Definition of Cooperative Verification . . . . .	7
<b>3</b>	<b>Literature Review(B.1)</b>	<b>9</b>
3.1	Methodology . . . . .	9
3.2	Classification of the Cooperative Verification Techniques . . . . .	10
3.3	Cooperative Actors and Artifacts . . . . .	13
<b>4</b>	<b>CoVeriTeam (B.2)</b>	<b>15</b>
4.1	Design . . . . .	16
4.2	Case Studies . . . . .	18
<b>5</b>	<b>Evaluation (B.3)</b>	<b>19</b>
5.1	Combinations of Verifiers . . . . .	19
5.2	Experiment setup . . . . .	20
5.3	Results . . . . .	22
5.4	Threats . . . . .	22
<b>6</b>	<b>CoVeriTeam Service: Verification as a Service(B.4)</b>	<b>25</b>
6.1	Service Design . . . . .	25
6.2	Use cases . . . . .	26
<b>7</b>	<b>Conclusion</b>	<b>27</b>
7.1	Critical Reflection . . . . .	27
7.2	Future Research . . . . .	27
7.3	Conclusion . . . . .	29

<b>Bibliography</b>	<b>31</b>
<b>A Credits</b>	<b>41</b>

<b>B</b>	<b>Original Manuscripts</b>	<b>43</b>
B.1	Cooperative Software Verification: An Exploratory Literature Review . .	44
B.2	CoVERITeam: On-Demand Composition of Cooperative Verification Sys- tems . . . . .	67
B.3	Construction of Verifier Combinations From Off-the-Shelf Components . .	86
B.4	CoVERITeam Webservice: Verification as a Service . . . . .	118



# List of Figures

2.1	A combination of a verifier and a validator . . . . .	5
2.2	Simplified classification hierarchy of verification actors (this figure is adapted from our article on CoVeriTeam [40]) . . . . .	7
2.3	Simplified classification hierarchy of verification artifacts (this figure is adapted from our article on CoVeriTeam [40]) . . . . .	7
3.1	Stages of the selection process and number of papers selected in each stage (this figure is taken from our literature review on cooperative verification [39])	10
3.2	Classes of identified cooperative verification techniques based on the use of artifacts (this figure is taken from our literature review on cooperative verification [39]) . . . . .	12
4.1	Types of combinations in CoVeriTeam (this figure is taken from our article on CoVeriTeam [40]) . . . . .	17
5.1	Combinations used in the experiments (this figure is taken from our article on the evaluation of combinations [49]) . . . . .	20
5.2	Construction of a <i>verifier+validator</i> combination (this figure is taken from our article on the evaluation of combinations [49]) . . . . .	21
5.3	Subsets of verification tools used in the combinations (this figure is taken from our article on the evaluation of combinations [49]) . . . . .	22
6.1	Abstract view of CoVeriTeam SERVICE (this figure is taken from our article on CoVeriTeam SERVICE [45]) . . . . .	25
6.2	Clients of CoVeriTeam SERVICE (this figure is taken from our article on CoVeriTeam SERVICE [45]) . . . . .	26



# 1 Introduction

Software is ubiquitous in our lives, and our dependence on it is increasing with each passing day. As a result, a bug or failure in software has the potential to have an adverse effect on us, ranging from monetary losses to loss of life [81, 100, 126]. These high stakes make the pursuit of bug-free software important. Software verification is used to prove that the software behaves as expected or to find bugs, allowing us to fix them in time.

Software verification has been an active area of research for decades [8, 19, 21, 64]. The research community has invented many techniques and developed numerous tools to tackle the problem [12, 13, 32, 54, 63, 73, 82, 85, 114, 115, 116]. Each of these techniques and tools handle a part of the problem. Some excel at solving particular kinds of specification, such as UAutomizer [94] for termination analysis or CPAChecker [92] and VeriAbs [88] for checking safety, while others extend support for more language features for a certain specification, such as Deagle [65] for concurrency or Symbiotic [89] and VeriAbs [88] for loops. The results of the international competition on software verification (SV-COMP) [20] over the past years show that even the best performing tools can solve only a subset of the used benchmark set [21], and there are considerable number of tasks that are solved by one tool but not by others [49]. Each of these techniques and tools have their own strengths and weaknesses.

It is imperative to combine the strengths of different verification tools and techniques in order to attempt to solve harder problems. The software verification community employs various approaches to combine verification techniques. These combinations could either be *integrated* or constructed using *off-the-shelf* components. In an integrated (also called white box) combination, the techniques are customized and adapted to fit together in a cohesive unit. A few examples of integrated combinations are: SMASH [69], Synergy [11], CPAChecker [32], Check 'N' Crash [14], and DyTA [68]. On the other hand, in an off-the-shelf (also called black box) combination, the components are used as they were, without any modification. A few examples of off-the-shelf combinations are: portfolios [10], algorithm selection [78], conditional testing [55], and MetaVal [35]. Off-the-shelf combinations can be further categorized as *non-cooperative* or *cooperative*.

In a non-cooperative off-the-shelf combination, the participating components do not exchange information. Portfolios and algorithm selection based combinations are non-cooperative off-the-shelf combinations. In a portfolio of verification tools, each tool is executed in isolation and the results of the one that succeeds in computing a result are considered, e.g., PredatorHP [104], CPAChecker [113], UFO [5]. Selection techniques



extract some features about the verification task (e.g., use of loops, arrays, etc.) and use them to choose a technique most suited to solve it [26, 78, 91, 109, 123].

In contrast to the above mentioned combinations, solvers would *team up* in a cooperative combination to solve the given verification problem. They would exchange information to help each other. Conditional verification [16, 38, 53, 55] and witness validation [27, 28, 29] are examples of cooperative verification.

In this thesis, we explore the field of cooperative verification. This exploration comprises making the notion of cooperative verification concrete, developing a framework to construct and execute such combinations, and evaluating tool combinations.

## 1.1 Contributions

We make the following research contributions:

1. Definition: We make the notion of cooperative verification concrete by giving a definition (Sections B.1 and B.2).
2. Literature review on cooperative verification [39]: We review scientific literature to provide an overview of cooperative verification techniques and develop a classification for them (Section B.1).
3. CoVeriTeam [40]: We present a conceptual framework and a tool for construction and execution of tool combinations (Section B.2).
4. Evaluation of tool combinations [49]: We present the results of an extensive evaluation of tool combinations to show that these combinations can perform better than the standalone tools (Section B.3).
5. CoVeriTeam Service [45]: We present a web service for CoVeriTeam allowing users to remotely execute verification tools and their combinations (Section B.4).

We developed the following tools during the course of this thesis:

1. CoVeriTeam [40, 47] is an open-source tool for construction and execution of tool combinations based on off-the-shelf components.
2. CoVeriTeam Service [45, 48] is an open-source web service that provides web-based access to CoVeriTeam allowing users to execute verification tools remotely.

## 1.2 Impact

Our work already has found use cases in the verification community:

1. CoVeriTeam was used in a research project on decomposing CEGAR [63] into off-the-shelf components. In this research, component-based CEGAR was constructed by combining off-the-shelf components using CoVeriTeam [24].
2. CoVeriTeam was used in a research project that generalizes the idea of *ranged symbolic execution* [75]. In this work, the path exploration is split into several ranges (sets of execution paths) and then these ranges are analyzed in parallel [77]. CoVeriTeam was used to orchestrate the execution of these analyses in parallel.

3. LIV [36] is a validator for safety proofs of programs written in C. It decomposes the validation problem into several smaller loop-free verification conditions expressed as programs in C. These smaller problems are then solved in parallel using off-the-shelf verifiers. CoVeriTeam is used to execute these off-the-shelf verifiers.
4. CoVeriTeam was used in the research work that constructs the unified framework for control-flow-based loop abstraction by Beyer, Rosenfeld, and Spiessl [33] using off-the-shelf components [34]. CoVeriTeam was used to provide interfaces to off-the-shelf verification tools.
5. Two tool combinations, a parallel portfolio of verifiers and a selection based verifier, created using CoVeriTeam were submitted to the international competition on software verification 2022 (SV-COMP) [21]. These tools performed very well in the competition; but were not ranked because the participation was *hors concours*.
6. GRAVES-PAR [121] combines the ideas of machine learning based selection and parallel portfolios. It was constructed using CoVeriTeam and submitted to SV-COMP 2023. It selects a set of verification tools to execute based on the given verification problem. First, it uses graph representation of the given program and employs a graph neural network to predict the resource consumption of the tools. Then, it selects a portfolio of verification tools based on this information and the given resource constraints. CoVeriTeam was used to execute the selected portfolio of verifiers.
7. CoVeriTeam SERVICE was used in the integration scripts for tool submission to SV-COMP and Test-Comp 2023. When a participating team submitted a tool, the integration scripts called CoVeriTeam SERVICE to check if the tool can be executed successfully or not. This automation saved manual effort of the organizer.
8. CoVeriTeam SERVICE has been adapted for the purpose of providing web based access to verification tools developed at Huawei Dresden Research Center. It is used for demonstrating the capabilities of these verification tools by the team, and allows other teams at Huawei to experiment with these tools. The service is accessible only internally at Huawei Technologies.

### 1.3 Structure

This thesis presents an overview of my PhD research. Chapter 2 defines cooperative verification and the notions of verification actors and artifacts. Chapter 3 gives a brief overview of the literature review on cooperative verification. Chapter 4 gives a brief introduction of CoVeriTeam, while Chapter 5 summarizes the experiments conducted to evaluate tool combinations. Chapter 6 provides an overview of CoVeriTeam SERVICE. Finally, Chapter 7 concludes this work with a critical reflection on our achievements and identifies directions for further research.



## 2 Definitions

Cooperative verification aims to harness the synergies of various verification techniques. A cooperative technique is constructed as an off-the-shelf combination of other verification techniques [23].

Cooperative verification requires various verification techniques to work together to solve a *verification problem*. The cooperating techniques cooperate through the means of information exchange. We use the term *verification actors* for producers and consumers of this information, and *verification artifacts* for the exchanged information. Examples of a verification problem are include a satisfiability check, test generation, feasibility check of a program path, and refinement check.

### 2.1 Example

Figure 2.1 shows a cooperative combination of a verifier and a validator. A verifier, e.g., a model checker like CPACHECKER or CBMC, takes a program and a specification as input and outputs if the program satisfies the specification or not. Additionally, it can output a witness—a justification of the verdict it produced. An example of such a witness is a counterexample trace reaching the error location if the verifier finds a bug.

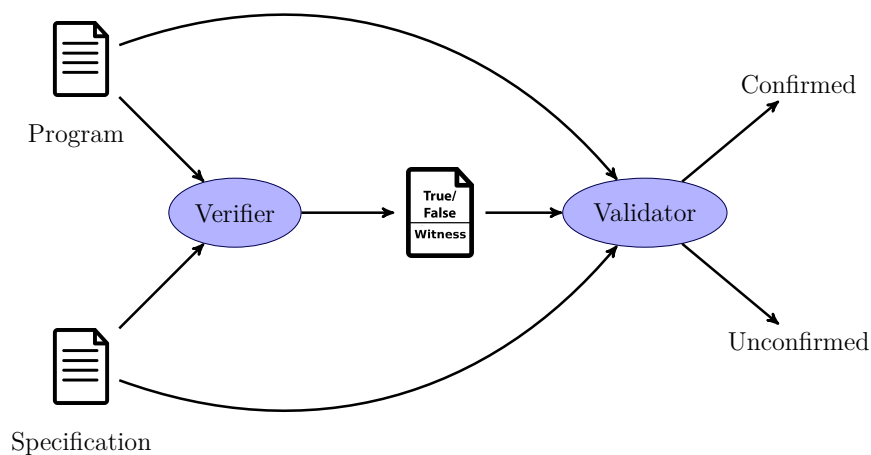


Figure 2.1: A combination of a verifier and a validator

Table 2.1: Some example of cooperative verification actors

Actor	Subtype Of	Description	Example Tools
Verifier	Analyzer	Tries to verify whether the given program satisfies the given input specification; produces a verdict and a witness justifying the verdict	CBMC [62], CPACHECKER [32], UAUTOMIZER [94]
Conditional Verifier	Analyzer	A verifier that also explicitly deals with a condition—a summary of the previously done verification work	CPACHECKER [32]
Tester	Analyzer	Takes a program and a coverage criterion, and generates test cases	KLEE [12]
Conditional Tester	Analyzer	Analogous to conditional model checker, a conditional tester explicitly considers test goals representing a subset of the total required work	CONDTEST [55]
Validator	Analyzer	Validates the results produced by a verifier	CPACHECKER [32], UAUTOMIZER [94]
Reducer	Transformer	Transforms a program with respect to a condition such that if the residual program satisfies the specification, so does the input program; uses a condition, i.e., a summary of the verification work already achieved, to <i>reduce</i> a program such that the already accomplished work need not be repeated	Reducer [38]
Instrumentor	Transformer	Instruments the program with information from a given artifact that can be hints from a witness, or labels applied to program locations deduced from a test criterion, etc.	METAVAL [35]

The verifier verifies the program and produces a verdict and a witness. The validator validates the output of the verifier. It tries to re-establish the verdict produced by the verifier with the help of the witness. The witness is *confirmed* if the validator is able to reach the same verdict, otherwise it is *unconfirmed*.

Verifier and validator are abstract interfaces for tools or techniques performing a specific function. We use the term *verification actors* for these interfaces. Witness is an example of the information passed between verification actors. We use the term *verification artifacts* for interfaces representing such information.

We now detail these notions and introduce the definition of cooperative verification.

## 2.2 Actors and Artifacts

**Actor.** A cooperative verification actor produces and/or consumes the information for cooperation. Verification actors are divided between *analyzers* and *transformers*. Analyzers produce new knowledge about the verification task, e.g., a verifier produces the knowledge if the property is satisfied or not. Transformers would transform the veri-

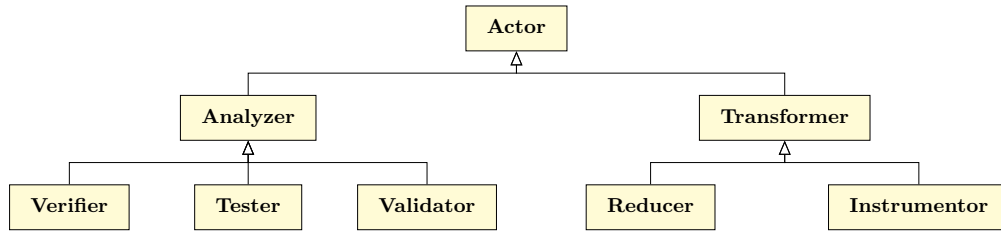


Figure 2.2: Simplified classification hierarchy of verification actors (this figure is adapted from our article on CoVeriTeam [40])

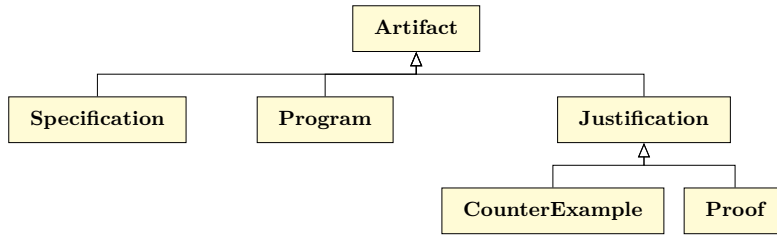


Figure 2.3: Simplified classification hierarchy of verification artifacts (this figure is adapted from our article on CoVeriTeam [40])

fication task, but not produce new knowledge, e.g., a tool transforming code from C to LLVM [87] such that a model checker for LLVM can be used for verification. Figure 2.2 shows a simplified classification hierarchy of cooperative verification actors, and Table 2.1 lists a few examples with explanations.

**Artifact.** Artifacts for cooperative verification contain information or knowledge about a verification problem. For example, a C program is an artifact representing a part of the verification problem. Figure 2.3 shows a simplified classification hierarchy of cooperative verification artifacts. Verification artifacts are divided between *specification*, *program*, and *justification*. Specifications include behavioral specifications for formal verification or for test generation. Programs include code written in various programming languages. Justifications are the artifacts supporting the result produced by analyzers, e.g., counterexample or proof. More details are available in the article on CoVeriTeam [40].

## 2.3 Definition of Cooperative Verification

**Definition 1.** A verification approach is called **cooperative**, if

1. *identifiable verification actors* pass information in form of
2. *identifiable verification artifacts* towards the common objective of
3. solving a verification problem,
4. where at least two of these actors are analyzers.

The above definition is taken from our article on literature review of cooperative verification techniques [39].

A cooperative verification technique is a combination of verification techniques, but not all combinations are cooperative. This definition excludes the following types of combinations:

1. Portfolio: A sequential or a parallel portfolio of verifiers is not cooperative because the verifiers do not share information with each other. Each verifier works alone for solving the verification task, and the result of the one that succeeds is considered.
2. Algorithm selection: Approaches where the first step is to select which algorithm or tool should be used to solve the given verification problem are not cooperative because the information is not shared. The selector selects which verifier to execute and that verifier is executed.
3. Syntactic transformation: Some verification combinations use preprocessing as a first step to convert the verification problem to a format such that an off-the-shelf verifier can be used to solve it. These types of approaches are excluded in our definition because these combinations use only one analyzer.

Our definition tries to formalize the idea that the techniques should cooperate, i.e., solve a part of the problem or produce some knowledge that could be helpful to some other technique for solving the problem. The cooperative verification techniques should be able to consume and/or produce additional information, other than the initial verification problem and the final result, about the verification task.

## 3 Literature Review (B.1)

Various literature reviews have been conducted in the area of software verification [6, 7, 22, 106, 107, 118]. However, prior to our work, a literature review focusing on cooperative verification had not been conducted. To address this gap, we conducted a review of scientific literature to provide an overview of the cooperative verification techniques and develop a classification for them.

We developed a methodology to conduct the literature review. During our review, we followed it to process articles published in leading scientific publications related to software verification. After following a multi-stage filtering process, we identified the verification techniques presented in 72 articles as cooperative. Our overview and classification is based on the findings from these 72 papers.

The article “Cooperative Software Verification: An Exploratory Literature Review” (Section B.1) reports on the findings of the conducted literature review.

### 3.1 Methodology

Figure 3.1 shows the methodology we followed for the literature review.

*Stage 0: publication selection.* Cooperative verification is related to the research fields of formal verification, software engineering, and semantics of programming languages. We selected A or A\* ranked conferences based on CORE rankings [18] in these fields. We then extended this list by a few other conferences that we consider important in the field. We considered publications by ACM or Springer.

In computer science, conferences are preferred over journals to publish new results [67, 97, 99]. Therefore, we excluded journals from our publication selection.

*Stage 1: selection of articles.* We considered the articles from 2012 onward from the publications selected in *Stage 0*. We selected articles with more than 7 pages for ACE/IEEE publications and more than 10 pages for Springer.

We got 8 108 papers to start the search after this stage of filtering.

*Stage 2: applying search query.* We applied a search query to the titles and abstracts of the articles output by *Stage 1*. The query was designed to filter articles related to verification, analysis, and testing.

After this stage of filtering, 4 332 papers remained in consideration.



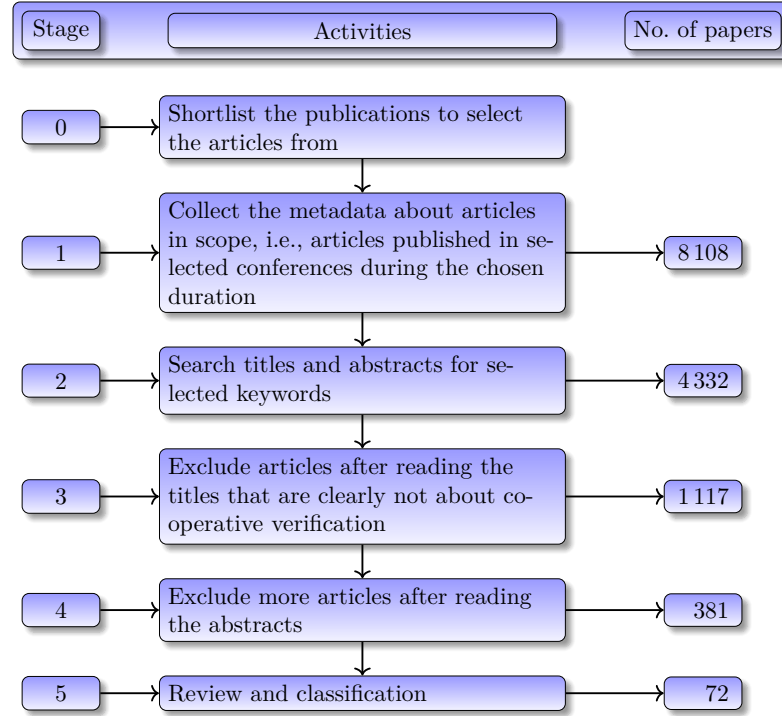


Figure 3.1: Stages of the selection process and number of papers selected in each stage (this figure is taken from our literature review on cooperative verification [39])

*Stage 3: exclusion based on titles.* We manually processed the titles of the papers in consideration. We filtered out the papers that we could confidently rule out as being cooperative. After this stage of filtering, 1 117 papers remained in consideration.

*Stage 4: exclusion based on abstracts.* We analyzed the abstracts of the remaining articles and excluded the ones which were not presenting a verification approach based on a combination of verification techniques.

As a result of this stage, 381 papers were left for review.

*Stage 5: review of selected articles.* We applied our definition of cooperative verification to the techniques presented in the 381 articles from the previous stage.

We identified 72 cooperative verification techniques.

## 3.2 Classification of the Cooperative Verification Techniques

We developed a classification for the verification techniques identified as cooperative. Figure 3.2 shows the class hierarchy of the identified classes, and Table 3.1 gives a brief overview of these classes. Our literature review [39] contains further details about the articles in each class.

### 3.2. CLASSIFICATION OF THE COOPERATIVE VERIFICATION TECHNIQUES

Table 3.1: Classification of the cooperative verification techniques (this table is taken from our literature review on cooperative verification [39])

Class name	Explanation	Examples	Count
Reduction	The verification task is reduced such that it can be solved by another analyzer.	[57, 101, 111]	12
Guide	The artifact produced by an analyzer acts to guide another analyzer.		
Conditional	First analyzer tries to solve the verification problem and produces an artifact that summarizes the work done; another actor then uses this information to focus only on the unsolved parts of the task.	[53, 55, 90]	12
Hint	An analyzer generates hints that are then used to guide the verification of another analyzer.	[74, 86, 120]	17
Scrutiny	The artifact produced is scrutinized by another analyzer.		
Validation	The result produced by one analyzer is validated by another analyzer.	[27, 28]	14
Refinement	The artifact produced by one analyzer is refined by another analyzer.	[1, 83]	3
Iterative Validation Guided Refinement	The artifact produced is first validated, and then the result of validation is used to guide the process of refinement. This sequence is repeated until a solution is found.	[24, 60]	14

#### Reduction

In this class of cooperative verification approaches, an analyzer reduces the verification problem such that it can be solved by another analyzer. Examples of such reductions include reducing a concurrent program into a sequential one such that the behavior of the two is the same with respect to the property under consideration [2, 72, 101], and a program to program transformation targeting loop behavior [57, 58]. We identified 12 articles in this class.

#### Guide

In this class of cooperative verification approaches, an analyzer produces knowledge that is used to guide the verification effort of a subsequent analyzer. This class is further divided into two classes: *Conditional* and *Hint*.

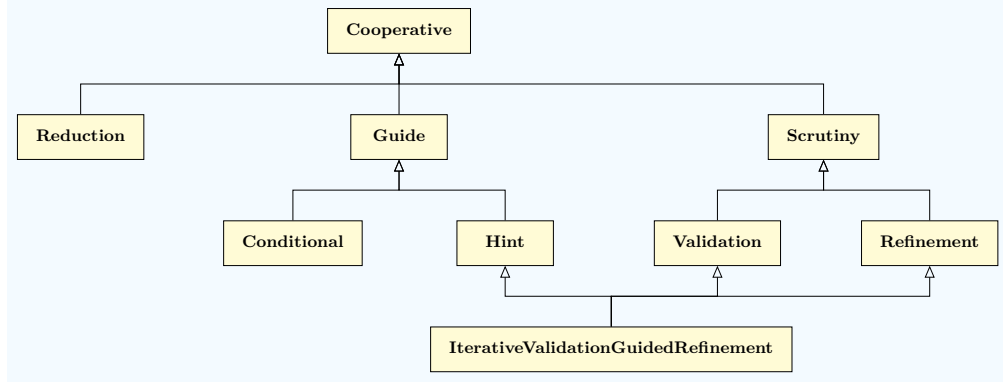


Figure 3.2: Classes of identified cooperative verification techniques based on the use of artifacts (this figure is taken from our literature review on cooperative verification [39])

**Conditional.** In conditional analyses, the first analyzer produces conditions under which the property under consideration holds. These conditions could be either a summary of the verification work already done [53, 55], or unsound assumptions made by an analyzer to progress the verification task [90], or partial typing information [66]. These conditions are used by a subsequent analyzer to reduce the verification effort by making use of the knowledge encoded in the conditions. We identified 12 articles in this class.

**Hint.** Some kinds of analysis of verification tasks can produce knowledge that might be useful for another analyzer as a hint for solving the task. Examples of such hints include a seed for fuzzing-based test generators [86, 120, 125], and invariants that can be injected in a verification procedure [30, 76]. We identified 17 articles in this class.

### Scrutiny

In this class of cooperative verification approaches, an analyzer scrutinizes the artifacts produced by another analyzer. The purpose of this scrutiny could be validation, filtering, or refinement of the given artifact. We further divide this class into two classes: *Validation* and *Refinement*.

**Validation.** The results of an unsound analyzer are not trustworthy. In these cases, it is desirable to validate the results produced by another analyzer. Examples of the artifacts requiring validation include alarms or warnings produced by a possibly unsound verifier [28, 79, 96], invariants or a ranking function produced by a machine learning based tool [70, 93], and a proof of correctness produced by an untrusted verifier [27]. We identified 14 articles in this class.

**Refinement.** An analyzer can also scrutinize the artifacts produced by another analyzer in order to refine them. Examples of such refinements include alarm refinement [83, 122], and test suite minimization [124]. We identified 3 articles in this class.

**Iterative Validation Guided Refinement.** In this class of cooperative techniques, first the result produced by an analyzer is scrutinized, and the result of scrutiny is used

to guide the refinement. This process is repeated until a solution is found. Examples of techniques in this class include construction of CEGAR based on off-the-shelf components [24], invariant synthesis based on a combination of a learner and a verifier [60, 103], and inference of loop bounds [9, 108]. We identified 14 articles in this class.

### 3.3 Cooperative Actors and Artifacts

We also found papers that presented a technique or a tool that can be used in a cooperative combination.

Examples of such actors include:

- Program transformers can produce an over-approximation or under-approximation of the given program, which can be given as input to an analyzer. For instance, Loop shrinking [112] transforms array processing loops in a program into loops with smaller bounds, over-approximating the program. An off-the-shelf model checker can be used to verify the transformed program.
- Validators scrutinize the output produced by a verifier, and as a result can confirm or reject it. One such tool is METAVAL [35], which converts a validation problem into a verification problem. It first instruments a witness in the program and then uses algorithm selection to select a background verifier.
- Invariant generators can be employed as cooperative actors in combination with techniques like  $k$ -induction [3]. For example, Dillig et al. [71] propose a technique for generating inductive invariants using a combination of verification condition generator and abductive inference.
- Refiners can refine the artifact produced by an analyzer. For example, test suite reduction refines the test suite by removing redundant test cases, thus producing a smaller test suite. A tool that reduces the given test suite [4, 110, 117] can be used in a cooperative combination.



## 4 CoVeriTeam (B.2)

The software verification research community employs various combinations of verification techniques [10, 15, 26, 51, 78, 84, 94, 95, 105, 109, 123], many of these are cooperative [24, 28, 37, 53, 55, 76, 90]. However, despite the interest of the verification community in developing tool combinations, the combinations were developed *ad hoc*: we were missing a framework to systematically construct these combinations.

To address this issue, we envisioned a solution based on interfaces for verification tools and a mechanism to combine these interfaces. Our solution uses the notions of *verification actors* and *verification artifacts* as defined in Section 2.2.

In our solution, each verification actor has two aspects: *descriptive* and *operational*. The descriptive aspect addresses conceptual questions about a verification actor, such as: *what kind of input does it consume?*, *what output does it produce?*, *what function does the actor perform on execution?*. The operational aspect deals with concerns regarding the execution of the actor, including: *what command should be executed?*, *executing tools in isolation to prevent interference with the host system*, and *controlling resource consumption*.

CoVeriTeam [40], the tool we developed, brings together both the descriptive and operational aspects of verification actors. It enables a systematic approach for constructing tool combinations and executing them. Using CoVeriTeam, a user can write a description for a verification actor in a simple language and execute it. CoVeriTeam is an open-source tool and publicly available under the Apache 2 license.

CoVeriTeam has already found use cases in the verification community, including: (1) its use in a modular implementation of CEGAR [24] based on off-the-shelf components, (2) a service based on CoVeriTeam (see Chapter 6) being used in the 2023 competitions on software verification and testing to test the execution of the submitted tool archives, and (3) its use in orchestrating the execution of different analyses in a generalization of ranged symbolic execution [75, 77].

We present CoVeriTeam in our article “CoVeriTeam: On-Demand Composition of Cooperative Verification Systems” (Section B.2). CoVeriTeam provides: (1) a language for describing combinations of verification actors, (2) an execution engine for executing these combinations, (3) an open-source implementation and a library of verification actors based on many well-known publicly available verification tools, (4) example case studies of combinations implemented in CoVeriTeam.

**Listing 1** Example of an actor definition file for the verifier CBMC [62]

---

```

1  actor_name: cbmc
2  toolinfo_module: "cbmc.py"
3  archives:
4    - version: default
5      location: "https://gitlab.com/.../cbmc.zip"
6      options: ['--graphml-witness', 'witness.graphml']
7  resourcelimits:
8    memlimit: "8 GB"
9    timelimit: "2 min"
10 format_version: '1.2'

```

---

## 4.1 Design

There are two main parts of the design of CoVeriTeam: description of actors and artifacts, and execution of actors.

### Actors and Artifacts

Actors in CoVeriTeam can be either *atomic*—based on a standalone tool, or *composite*—based on a combination of multiple actors.

**Atomic Actors.** In CoVeriTeam, atomic actors are based on tool archives. For example, an actor based on a model checker like CBMC or CPACHECKER would be an atomic actor in CoVeriTeam. When executed, an atomic actor triggers the execution of the underlying tool.

Lets consider a use case in which a user wants to verify a C program using the verifier CBMC. To achieve this, the user needs to have CBMC installed, construct the command, execute it, and understand its output. Similarly, CoVeriTeam requires the following to execute an atomic actor and extract the produced artifacts: the tool archive, the command to invoke the tool, and a function to process the tool output and extract artifacts. Additionally, we enforce resource limitations on the execution of tools and execute them in isolation.

To fulfill some of these requirements, we use features from BENCHEXEC [52]. We use tool-info modules from BENCHEXEC to assemble the command and process the tool output, and RUNEXEC to execute the tools in isolation and enforce resource limitations. A tool-info module is a few lines of Python code containing functions to create the command to be executed and to process the output produced by the tool. RUNEXEC uses control groups and namespaces features from the Linux kernel to measure and control the resource consumption and execute tools in isolation.

CoVeriTeam uses YAML configuration files, called actor definitions, to configure the atomic actors. Listing 1 shows an example of such an actor definition for the verification tool CBMC. It specifies the tool-info module to be used for executing this actor, the parameters to be passed to it, and resource limitations to be enforced. CoVeriTeam uses

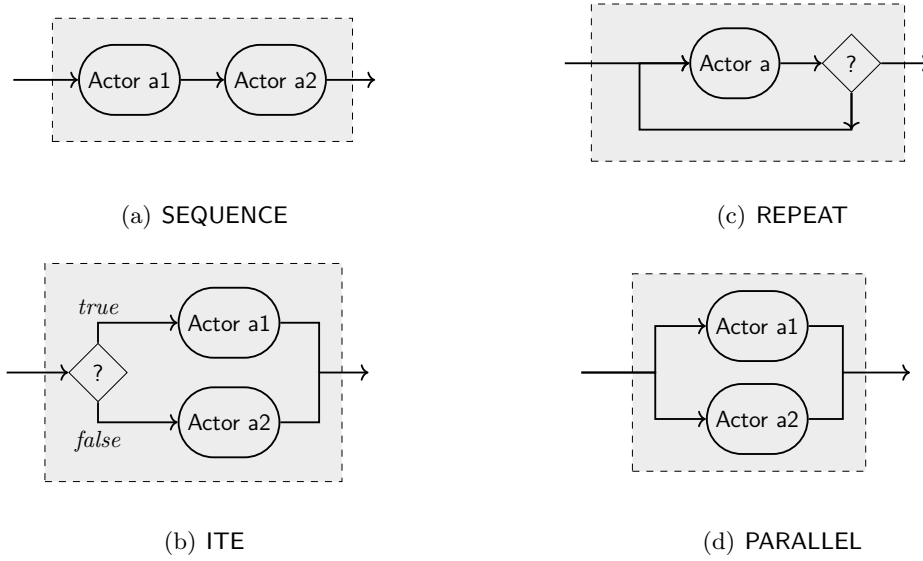


Figure 4.1: Types of combinations in CoVeriTeam (this figure is taken from our article on CoVeriTeam [40])

this information to download the tool, assemble the command, execute the command, and process the output produced by the execution.

**Composite Actors.** Verification actors can be combined to create a composite actor. The combination of a verifier and a validator presented in Fig. 2.1 is an example of a composite actor. Figure 4.1 shows the combinations available in CoVeriTeam: **SEQUENCE** executes the actors in sequence, providing the artifacts generated by the first one as input to the second one; **PARALLEL** executes the actors in parallel, waiting for each actor to finish execution; **ITE** chooses the actor to execute based on the value of a condition; **REPEAT** executes the actor repeatedly until the termination condition is satisfied. The article on CoVeriTeam contains more details on these combinations [40].

In addition to these combinations, we have also developed **PARALLEL-PORTFOLIO** combination. Actors in a **PARALLEL-PORTFOLIO** are executed until one of them produces an acceptable result, and then all the remaining actor executions are terminated. We have presented this combination in the article on evaluation [43, 49].

**Artifacts.** The artifacts in CoVeriTeam are based on files or strings. For example, an artifact for a C program would contain the path to the underlying C file, a test suite would contain the path of the directory containing test cases, a verdict would be a string like *SUCCESS* or *FAIL*.

## Execution

We have developed a simple domain specific language to describe creation of actors and artifacts. A user needs to provide this description file as input to CoVeriTeam. CoVeriTeam then creates the actors and their combination as described in the input



Table 4.1: Tool combinations implemented in CoVeriTeam (this table is adapted from our article on CoVeriTeam [40])

Technique	Year	Case Study	More Info
Witness Validation [27, 28]	2015, 2016	✓	<a href="#">Section B.2</a>
Execution-Based Validation [31]	2018	✓	<a href="#">More info</a>
Reducer [38]	2018	✓	<a href="#">More info</a>
CONDTEST [55]	2019	✓	<a href="#">More info</a>
METAVAL [35]	2020	✓	<a href="#">More info</a>

description file, and executes them on the provided artifacts. The execution engine of CoVeriTeam orchestrates the execution of the combinations managing the flow of artifacts between actors; and prepares commands for the execution of atomic actors and delegates the execution of these commands to RUNEXEC.

## 4.2 Case Studies

We have implemented some tool combinations and cooperative verification techniques in CoVeriTeam demonstrating that our tool can be used to create and execute tool combinations. [Table 4.1](#) lists these examples and case studies.

## 5 Evaluation (B.3)

Software verification tools specialize in solving specific types of verification problems. This specialization can be related to the specification, such as reachability analysis or termination analysis, or to the features of the program, like the usage of unbounded loops, arrays, floats, etc. The results of the software verification competition [20] in the past years consistently confirm this observation.

In our study, we investigate the effectiveness of tool combinations in comparison to standalone tools. We conducted extensive experiments to determine if the combinations of verifiers in a sequential portfolio, a parallel portfolio [10], or using an algorithm selection mechanism [78] could yield better results than any of the standalone verifiers. We used CoVeriTeam to systematically construct combinations of verification tools and evaluated them on a large benchmark set to find the answer.

We present the details of this construction and results of the experimental evaluation in the article “Construction of Verifier Combinations From Off-the-Shelf Components” (Section B.3). This article is an extended version of the article “Construction of Verifier Combinations Based on Off-the-Shelf Verifiers” [43].

### 5.1 Combinations of Verifiers

We evaluated the following three types of verifier combinations (Figure 5.1):

1. Sequential portfolio of verifiers: Verifiers are executed one after another in sequence until one of them produces a result. If a verifier is not able to produce a result within the allotted time, it is terminated and the next verifier is executed. Available CPU time is equally divided between the verifiers.
2. Parallel portfolio of verifiers: All verifiers run simultaneously until one of them produces a result that meets the termination condition. The remaining verifiers are then terminated. Available memory is equally divided between the verifiers.
3. Algorithm selection: First, a selection algorithm is executed to determine which verifier to use. Then, all available resources are allocated to that verifier. In our combination, we used a machine learning based algorithm selector [109].

We used CoVeriTeam to construct these combinations using off-the-shelf components.

In our previous experiments [43], we found that both sequential and parallel portfolios produced more incorrect results in contrast to the standalone tools and algorithm

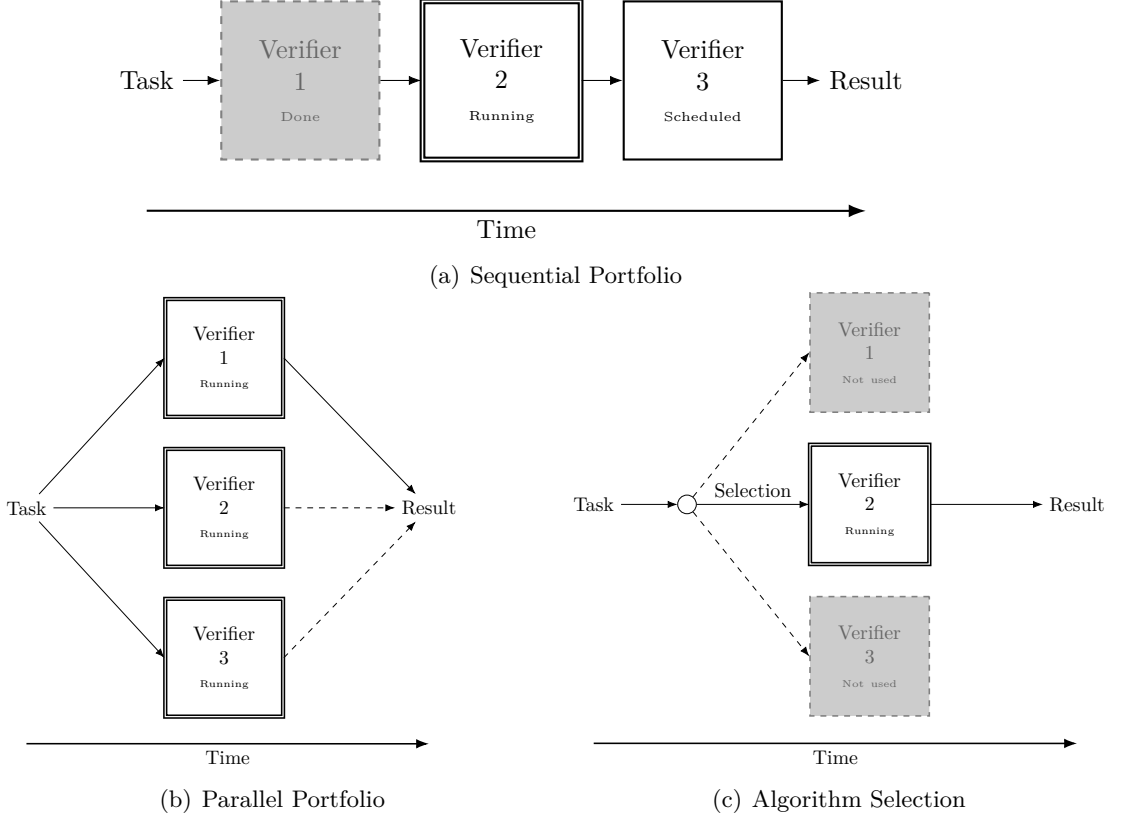


Figure 5.1: Combinations used in the experiments (this figure is taken from our article on the evaluation of combinations [49])

selection. The portfolios are biased towards fast tools, and coincidentally there was a tool in our combinations that was more unsound than others and produced results fast.<sup>1</sup> To address this issue, we included a validation step in our experiments. The validation step executes a parallel portfolio of validators (as shown in Fig. 5.2). This combination serves as a verifier itself. We combine this construction of *verifier+validator* based on different verifiers in sequential portfolio, parallel portfolio, and algorithm selection.

## 5.2 Experiment setup

*Selection of Verifiers.* We selected the verifiers for our combinations based on the results of the 11th software verification competition (2022) [21]. We sorted the best 8 tools from the REACHSAFETY category based on their scores in SV-COMP 2022. For creating combinations of  $n$  tools, we chose the top  $n$  tools from this sorted list. Figure 5.3 illustrates

<sup>1</sup>If the unsound tool had also been inefficient in terms of resource consumption, the portfolios would have produced fewer incorrect results.

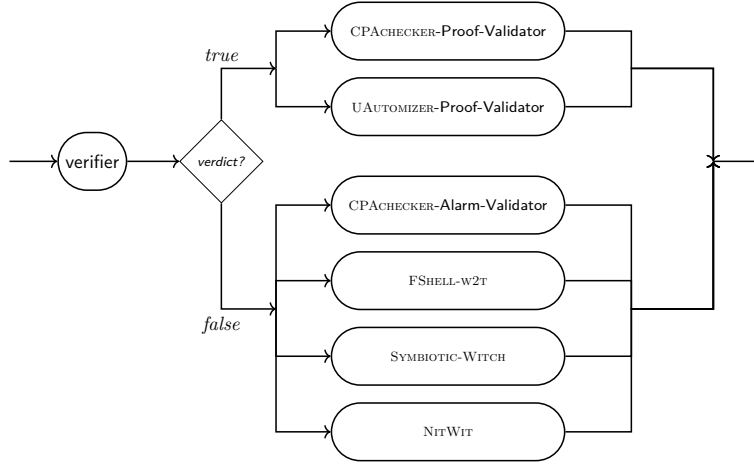


Figure 5.2: Construction of a *verifier+validator* combination (this figure is taken from our article on the evaluation of combinations [49])

the sets of verifiers that we used in different types of combinations. We selected the following verifiers: CPACHECKER [92], ESBMC [98], SYMBIOTIC [89], UAUTOMIZER [94], CBMC [59], PINAKA [61], UTAIPAN [56], 2LS [119].

*Selection of Validators.* Similar to the verifiers, we chose the validators based on the results of the 11th software verification competition (2022) [21]. We chose the most effective validators for confirming witnesses in the competition as reported in [25]. For violation witness validation (*alarm validation*), we chose four validators: CPACHECKER-based violation witness validator [28], FSHELL-W2T [31], SYMBIOTIC-WITCH [102], and NITWIT [80]. For the validation of proofs, we chose two correctness witness validators (*proof validators*): CPACHECKER-based and UAUTOMIZER-based correctness witness validators [27].

*Benchmark Set.* For our experiments, we considered all the verification tasks that were used in the software verification competition 2022 in the competition category titled REACHSAFETY. Each verification task consists of a program written in C and a specification. The specification for our experiments is a safety property stating that an error location should never be reached. The complete benchmark set is publicly available [17]. The benchmark set for our experiments consisted of 5 400 verification tasks in total.

*Execution Environment.* We executed the experiments on the same infrastructure and with same resource limits as SV-COMP 2022. The details are:

- *Machine configuration:* one 3.4 GHz CPU (IntelXeon E3-1230 v5) with 8 processing units (virtual cores), 33 GB RAM, Ubuntu 20.04 operating system
- *Resource limits:* 8 processing units, 15 min of CPU time, and 15 GB memory for each verification run

*Benchmark Execution.* We used the state-of-the-art benchmarking framework BENCHEXEC [52] for executing our benchmarks. We computed the scores and data for our plots based on the measurements generated by BENCHEXEC. We used the same scoring schema as used by SV-COMP [21].

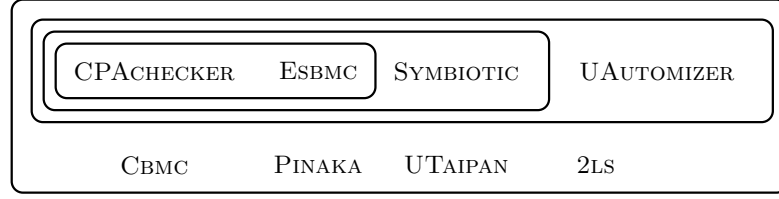


Figure 5.3: Subsets of verification tools used in the combinations (this figure is taken from our article on the evaluation of combinations [49])

### 5.3 Results

The results of our experiments show that each combination has a configuration that performs better than the best standalone tool.

However, the effectiveness of portfolios starts to decrease as the number of tools in the portfolio increases. As the number of tools grows and the total available resources remain constant, the resources available to each tool decrease. This, in turn, means that at some point, a verification tool does not get enough resources to solve non-trivial tasks.

Algorithm selection, on the other hand, performs better as the number of available tools to select from increases. As the resources are not shared between the verifiers, the chosen verifier gets all the available resources, irrespective of the number of verifiers available. This increases the available choices without sacrificing resource availability.

Introducing a validation step makes portfolios more competitive. Our findings in the previous experiments [43] showed that the presence of an unsound but fast tool can have a strong adverse effect on the qualitative performance of portfolios by making them produce a higher number of incorrect results. This is because portfolios are biased towards tools that finish fast (for sequential portfolio the position of the tool in the sequence is also important). Using validators mitigates this issue and relieves the user from the worry of choosing only those tools that have a very low rate of incorrect results.

### 5.4 Threats

Our experiments show that tool combinations can improve the effectiveness of verifiers, but this does not imply that tool combinations always improve performance. The results of our experiments are strongly influenced by the setup of our experiments. Keeping these factors in consideration can help a user design combinations that perform better.

*Resource Availability and Complexity of Tasks.* Our experiments benefited from the fact that the benchmark set was diversified, and the resource limitations were generous. The portfolios were helped by the fact that there were sets of tasks that were difficult for one verifier but easy for another. In a portfolio, by its nature, each of the verifiers is allocated fewer resources than any standalone tool. However, there was some verifier in the portfolio that succeeded in producing a result even with the reduced resources.

Algorithm selection exploited this situation even further by selecting the best suited verifier and giving it all the available resources.

*Knowledge about the Verification Task.* The choice of our tools was based on their performance on the same benchmark set. This was even more pronounced with the machine learning based algorithm selector. If a user is given a set of verification tasks about which there is no knowledge, then it would be difficult to create a winning combination. In this case, the user still has to make some assumptions about the given verification task and decide whether to use a standalone tool or a combination based on these assumptions.



## 6 CoVeriTeam Service: Verification as a Service (B.4)

CoVeriTeam provides an interface for verification actors and allows users to create and execute combinations of these actors. However, CoVeriTeam executes the verification tools and their combinations on the machine of the user.

Some users might prefer to avoid executing a verification tool on their machine. Some of them might have security concerns about a tool downloaded from an untrusted location. The local machine might not be powerful enough to execute a computationally heavy tool. There could be a mismatch between the available and required system configuration. Some verifiers might require permissions that the user does not have.

We addressed these challenges to make verification tools more accessible by developing a web service for CoVeriTeam. The article “CoVeriTeam Webservice: Verification as a Service” (Section B.4) introduces this idea.

### 6.1 Service Design

Figure 6.1 shows an abstract view of CoVeriTeam SERVICE. The service is designed as a REST API. To use the service, a user has to send the actor definitions, description

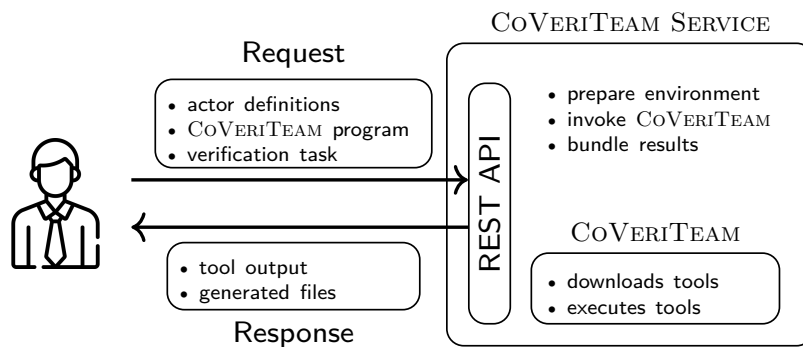


Figure 6.1: Abstract view of CoVeriTeam SERVICE (this figure is taken from our article on CoVeriTeam SERVICE [45])



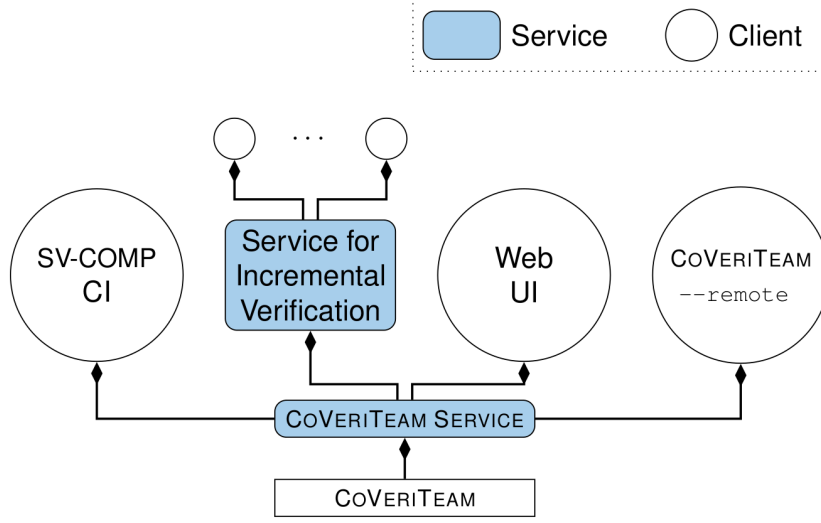


Figure 6.2: Clients of CoVeriTeam SERVICE (this figure is taken from our article on CoVeriTeam SERVICE [45])

of the combination to execute, and the verification task to the service. CoVeriTeam SERVICE then prepares the environment for execution, prepares the command and invokes CoVeriTeam, packages the results produced by CoVeriTeam, and sends them back to the user. The user receives the log and artifact files generated by the execution of tools or their combinations.

## 6.2 Use cases

CoVeriTeam SERVICE can be called from CoVeriTeam by appending the `--remote` flag to a CoVeriTeam command or by using a cURL request.

CoVeriTeam SERVICE has been used in the continuous integration process of the 2023 competitions on software verification (SV-COMP) and testing (Test-Comp). It is called upon submission of a tool to test if the tool can be successfully executed. Furthermore, we developed a web user interface to execute verification tools participating in SV-COMP 2023. Figure 6.2 shows some of the clients of the CoVeriTeam SERVICE.

CoVeriTeam SERVICE allows users to execute verification tools remotely and benefits tool developers by making it easier for them to provide access to their verification tools. Additionally, development of a generic API like this one for verification tools may facilitate commercial companies in integrating their tools as a service.

CoVeriTeam SERVICE is developed as an open-source project [48], and we also provide instructions to host an instance. The service is available for experimentation at <https://coveriteam-service.sosy-lab.org>.

## 7 Conclusion

### 7.1 Critical Reflection

There are two orthogonal aspects of cooperative verification: 1) decomposition of a problem into smaller problems and solving them with the best suited solvers, and 2) orchestration of the execution of these specialized solvers, enabling them to exchange knowledge.

The major part of this thesis focuses on the second aspect, i.e., developing a mechanism to enable various cooperative actors to cooperate. This mechanism is essential for cooperative verification, but it assumes the availability of cooperative actors and artifacts required by them. The full potential of cooperative verification can only be achieved with more research on problem decomposition and the development of specialized solvers.

Moreover, CoVeriTeam, our tool, currently supports combinations of only those verification actors that run to completion and can be executed on the same machine. This limits the combinations available for experimentation. For example, we cannot currently construct and execute a combination in CoVeriTeam that is based on two actors that require different system configurations.

Additionally, the effectiveness of cooperative verification techniques would be inconclusive if evaluated solely on academic benchmarks. We cannot arrive at a definitive conclusion unless we evaluate these techniques on more complex real-world problems.

### 7.2 Future Research

#### Policies for Cooperation

Up until now, our focus has been on studying cooperation between tools and developing a framework to construct tool combinations and orchestrate their execution. We assumed that the user is allowed or willing to execute all available tools.

However, a user might have preferences or be bound by some constraints about the tools to execute. We need a mechanism to express these constraints in the form of a *policy*. Once the user has specified a policy, we can check if the given policy allows a combination of tools to be executed. Examples of such constraints include the source of the tool archive, system configuration required by the actor, permitted licenses, etc. CoVeriTeam can be extended to support specification and checking of such policies.

### Provisioning Environment

Currently, CoVeriTeam supports execution only on the host machine. However, the system configuration of a user may not always support execution of a given tool archive. CoVeriTeam Service partially addresses this issue by allowing a user to execute a tool remotely, but it still requires the server to match the required system configuration.

To address this issue, we can extend CoVeriTeam to support execution of tools in DOCKER (or PODMAN) containers. A user can specify the container image providing the required execution environment, and CoVeriTeam can provision a container based on the image and run the given verification tool in it.

### Concurrently Communicating Actors

At present, CoVeriTeam is limited to cooperation between tools that *run-to-completion*. The next step is to extend it to support concurrently executing actors that can communicate with each other. Also, we need to investigate which architecture design is better suited for this kind of cooperation. Two possibilities are: a service oriented architecture or a publisher-subscriber model. This improvement would open opportunities for the design of more flexible systems employing various cooperative verification techniques making use of large distributed infrastructure.

### Porting Ideas from Model Based Software Engineering

The design of CoVeriTeam is inspired by model-based software engineering. At the core of CoVeriTeam is a component model that serves as the basis for its domain specific language, combinations, and their execution. More ideas from the area of model-based software engineering can be integrated in CoVeriTeam such as generating diagrams of CoVeriTeam combinations and providing support for a graphical language to create combinations. Some of these features would improve usability, whereas others could strengthen the theoretical aspects of cooperation.

### Artifacts for Cooperation

Cooperative verification actors achieve cooperation by using artifacts. For cooperation to be successful, cooperating actors must agree on the content and format of the information they exchange. Our research has identified the following artifacts as the most commonly used for cooperation: witnesses for alarms, invariants, (partial) proofs, program paths, and program slices. However, we expect that there are many more artifacts that could be identified for cooperation. Further research is necessary to reach a consensus on what knowledge is valuable to exchange and in what format.

### Standardization of Interfaces for Cooperation

Cooperation in verification requires common information exchange formats and interfaces for verification tools. Our work identifies and classifies many existing information

exchange formats and tools. However, further work is needed to identify additional cooperative actors and artifacts. Once we have a significant corpus of such actors and artifacts, we can move towards developing a standardization of interfaces for cooperation. Such standardization will help to integrate these tools across the industry, making cooperative verification more accessible in practice.

### 7.3 Conclusion

This research explores the field of cooperative verification from the perspective of software engineering. We view verification tools as interfaces and investigate and address the challenges arising from this perspective.

(1) We systematically reviewed the scientific literature to provide an overview and develop a classification of cooperative verification techniques.

(2) We developed a conceptual framework, supported by the tool `CoVeriTeam`, providing interfaces for cooperation through the means of various cooperative verification actors and artifacts. `CoVeriTeam` allows researchers to create and execute combinations of actors enabling them to experiment with cooperative verification techniques.

(3) We conducted large-scale experiments to demonstrate the effectiveness of tool combinations compared to standalone tools.

(4) We developed `CoVeriTeam Service`, a web service built on top of `CoVeriTeam` that allows users to remotely execute a verification tool.

Our work systematizes knowledge about cooperative verification, enables experimentation with combinations of verification actors, and creates opportunities for further research in the area of cooperative verification. `CoVeriTeam`, the tool developed as part of this thesis, has already been used and accepted in the verification community.

Our research begins the process of standardizing interfaces for cooperation between verification tools. This step is crucial in bringing cooperative verification to practice.

#### Availability

All of the papers in the scope of this thesis are (or will be) published as *open access*. Both `CoVeriTeam` and `CoVeriTeam Service` are open source projects; their code is available on GitLab [47, 48]. Additionally, we have uploaded the reproduction packages for the articles on Zenodo for long term availability [41, 42, 44, 46, 50].



# Bibliography

- [1] A. Albarghouthi, A. Gurfinkel, and M. Chechik. “From Under-Approximations to Over-Approximations and Back”. In: *Proc. TACAS*. LNCS 7214. Springer, 2012, pp. 157–172. DOI: [10.1007/978-3-642-28756-5\\_12](https://doi.org/10.1007/978-3-642-28756-5_12).
- [2] A. Dan, Y. Meshman, M. Vechev, and E. Yahav. “Effective Abstractions for Verification under Relaxed Memory Models”. In: *Proc. VMCAI*. Springer, 2015, pp. 449–466. ISBN: 9783662460801, 9783662460818. DOI: [10.1007/978-3-662-46081-8\\_25](https://doi.org/10.1007/978-3-662-46081-8_25).
- [3] A. F. Donaldson, L. Haller, D. Kröning, and P. Rümmer. “Software Verification Using k-Induction”. In: *Proc. SAS*. LNCS 6887. Springer, 2011, pp. 351–368. DOI: [10.1007/978-3-642-23702-7\\_26](https://doi.org/10.1007/978-3-642-23702-7_26).
- [4] A. Gotlieb and D. Marijan. “FLOWER: optimal test suite reduction as a network maximum flow”. In: *Proc. ISSTA*. ACM, 2014. DOI: [10.1145/2610384.2610416](https://doi.org/10.1145/2610384.2610416).
- [5] A. Gurfinkel, A. Albarghouthi, S. Chaki, Y. Li, and M. Chechik. “UFO: Verification with Interpolants and Abstract Interpretation (Competition Contribution)”. In: *Proc. TACAS*. LNCS 7795. Springer, 2013, pp. 637–640. DOI: [10.1007/978-3-642-36742-7\\_52](https://doi.org/10.1007/978-3-642-36742-7_52).
- [6] A. K. Karna, Y. Chen, H. Yu, H. Zhong, and J. Zhao. “The role of model checking in software engineering”. In: *Frontiers Comput. Sci.* 12.4 (2018), pp. 642–668. DOI: [10.1007/s11704-016-6192-0](https://doi.org/10.1007/s11704-016-6192-0).
- [7] A. P. Kaleeswaran, A. Nordmann, T. Vogel, and L. Grunske. “A systematic literature review on counterexample explanation”. In: *Information and Software Technology* 145 (2022), p. 106800. ISSN: 0950-5849. DOI: [10.1016/j.infsof.2021.106800](https://doi.org/10.1016/j.infsof.2021.106800).
- [8] A. Turing. “Checking a Large Routine”. In: *Report on a Conference on High Speed Automatic Calculating Machines*. <https://turingarchive.kings.cam.ac.uk/publications-lectures-and-talks-amtb/amt-b-8>. Cambridge Univ. Math. Lab., 1949, pp. 67–69.
- [9] A. V. Nori and R. Sharma. “Termination proofs from tests”. In: *Proc. ESEC/FSE*. ACM, 2013. DOI: [10.1145/2491411.2491413](https://doi.org/10.1145/2491411.2491413).

- [10] B. A. Huberman, R. M. Lukose, and T. Hogg. “An Economics Approach to Hard Computational Problems”. In: *Science* 275.7 (1997), pp. 51–54. DOI: [10.1126/science.275.5296.51](https://doi.org/10.1126/science.275.5296.51).
- [11] 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*. ACM, 2006, pp. 117–127. DOI: [10.1145/1181775.1181790](https://doi.org/10.1145/1181775.1181790).
- [12] C. Cadar, D. Dunbar, and D. R. Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *Proc. OSDI*. USENIX Association, 2008, pp. 209–224.
- [13] 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. Springer, 2015, pp. 3–11. DOI: [10.1007/978-3-319-17524-9\\_1](https://doi.org/10.1007/978-3-319-17524-9_1).
- [14] C. Csallner and Y. Smaragdakis. “Check ’n’ crash: Combining static checking and testing”. In: *Proc. ICSE*. ACM, 2005, pp. 422–431. DOI: [10.1145/1062455.1062533](https://doi.org/10.1145/1062455.1062533).
- [15] C. Richter, E. Hüllermeier, M.-C. Jakobs, and H. Wehrheim. “Algorithm selection for software validation based on graph kernels”. In: *Autom. Softw. Eng.* 27.1 (2020), pp. 153–186. DOI: [10.1007/s10515-020-00270-x](https://doi.org/10.1007/s10515-020-00270-x).
- [16] M. Christakis, P. Müller, and V. Wüstholtz. “Collaborative Verification and Testing with Explicit Assumptions”. In: *Proc. FM*. LNCS 7436. Springer, 2012, pp. 132–146. DOI: [10.1007/978-3-642-32759-9\\_13](https://doi.org/10.1007/978-3-642-32759-9_13).
- [17] *Collection of Verification Tasks*. <https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/>. Accessed: 2023-03-10.
- [18] *Computing Research & Education Conference Portal*. <http://portal.core.edu.au/conf-ranks/>. Accessed: 2023-04-01.
- [19] P. Cousot and R. Cousot. “Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints”. In: *Proc. POPL*. ACM, 1977, pp. 238–252.
- [20] D. Beyer. *Competition on Software Verification (SV-COMP)*. <https://sv-comp.sosy-lab.org/>. Accessed: 2023-03-10.
- [21] D. Beyer. “Progress on Software Verification: SV-COMP 2022”. In: *Proc. TACAS (2)*. LNCS 13244. Springer, 2022, pp. 375–402. DOI: [10.1007/978-3-030-99527-0\\_20](https://doi.org/10.1007/978-3-030-99527-0_20).
- [22] D. Beyer and A. Podelski. “Software Model Checking: 20 Years and Beyond”. In: *Principles of Systems Design*. LNCS 13660. Springer, 2022, pp. 554–582. DOI: [10.1007/978-3-031-22337-2\\_27](https://doi.org/10.1007/978-3-031-22337-2_27).
- [23] D. Beyer and H. Wehrheim. “Verification Artifacts in Cooperative Verification: Survey and Unifying Component Framework”. In: *Proc. ISoLA (1)*. LNCS 12476. Springer, 2020, pp. 143–167. DOI: [10.1007/978-3-030-61362-4\\_8](https://doi.org/10.1007/978-3-030-61362-4_8).

- [24] D. Beyer, J. Haltermann, T. Lemberger, and H. Wehrheim. “Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR”. In: *Proc. ICSE*. ACM, 2022, pp. 536–548. DOI: [10.1145/3510003.3510064](https://doi.org/10.1145/3510003.3510064).
- [25] D. Beyer and J. Strejček. “Case Study on Verification-Witness Validators: Where We Are and Where We Go”. In: *Proc. SAS*. LNCS 13790. Springer, 2022, pp. 160–174. DOI: [10.1007/978-3-031-22308-2\\_8](https://doi.org/10.1007/978-3-031-22308-2_8).
- [26] D. Beyer and M. Dangl. “Strategy Selection for Software Verification Based on Boolean Features: A Simple but Effective Approach”. In: *Proc. ISoLA*. LNCS 11245. Springer, 2018, pp. 144–159. DOI: [10.1007/978-3-030-03421-4\\_11](https://doi.org/10.1007/978-3-030-03421-4_11).
- [27] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. “Correctness Witnesses: Exchanging Verification Results Between Verifiers”. In: *Proc. FSE*. ACM, 2016, pp. 326–337. DOI: [10.1145/2950290.2950351](https://doi.org/10.1145/2950290.2950351).
- [28] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. “Witness Validation and Stepwise Testification across Software Verifiers”. In: *Proc. FSE*. ACM, 2015, pp. 721–733. DOI: [10.1145/2786805.2786867](https://doi.org/10.1145/2786805.2786867).
- [29] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, T. Lemberger, and M. Tautschnig. “Verification Witnesses”. In: *ACM Trans. Softw. Eng. Methodol.* 31.4 (2022), 57:1–57:69. DOI: [10.1145/3477579](https://doi.org/10.1145/3477579).
- [30] D. Beyer, M. Dangl, and P. Wendler. “Boosting k-Induction with Continuously-Refined Invariants”. In: *Proc. CAV*. LNCS 9206. Springer, 2015, pp. 622–640. DOI: [10.1007/978-3-319-21690-4\\_42](https://doi.org/10.1007/978-3-319-21690-4_42).
- [31] D. Beyer, M. Dangl, T. Lemberger, and M. Tautschnig. “Tests from Witnesses: Execution-Based Validation of Verification Results”. In: *Proc. TAP*. LNCS 10889. Springer, 2018, pp. 3–23. DOI: [10.1007/978-3-319-92994-1\\_1](https://doi.org/10.1007/978-3-319-92994-1_1).
- [32] D. Beyer and M. E. Keremoglu. “CPACHECKER: A Tool for Configurable Software Verification”. In: *Proc. CAV*. LNCS 6806. Springer, 2011, pp. 184–190. DOI: [10.1007/978-3-642-22110-1\\_16](https://doi.org/10.1007/978-3-642-22110-1_16).
- [33] D. Beyer, M. L. Rosenfeld, and M. Spiessl. “A Unifying Approach for Control-Flow-Based Loop Abstraction”. In: *Proc. SEFM*. LNCS 13550. Springer, 2022, pp. 3–19. DOI: [10.1007/978-3-031-17108-6\\_1](https://doi.org/10.1007/978-3-031-17108-6_1).
- [34] D. Beyer, M. L. Rosenfeld, and M. Spiessl. “CEGAR-PT: A Tool for Abstraction by Program Transformation”. In: unpublished manuscript, 2023.
- [35] D. Beyer and M. Spiessl. “METAVAL: Witness Validation via Verification”. In: *Proc. CAV*. LNCS 12225. Springer, 2020, pp. 165–177. DOI: [10.1007/978-3-030-53291-8\\_10](https://doi.org/10.1007/978-3-030-53291-8_10).
- [36] D. Beyer and M. Spiessl. *Source-Code Repository of LIV*. <https://gitlab.com/sosy-lab/software/liv>. Accessed: 2023-03-07.



- [37] D. Beyer and M.-C. Jakobs. “CoVeriTest: Cooperative Verifier-Based Testing”. In: *Proc. FASE*. LNCS 11424. Springer, 2019, pp. 389–408. DOI: [10.1007/978-3-030-16722-6\\_23](https://doi.org/10.1007/978-3-030-16722-6_23).
- [38] D. Beyer, M.-C. Jakobs, T. Lemberger, and H. Wehrheim. “Reducer-Based Construction of Conditional Verifiers”. In: *Proc. ICSE*. ACM, 2018, pp. 1182–1193. DOI: [10.1145/3180155.3180259](https://doi.org/10.1145/3180155.3180259).
- [39] D. Beyer and S. Kanav. “Cooperative Software Verification: An Exploratory Literature Review”. In: unpublished manuscript, 2025.
- [40] D. Beyer and S. Kanav. “CoVeriTeam: On-Demand Composition of Cooperative Verification Systems”. In: *Proc. TACAS*. LNCS 13243. Springer, 2022, pp. 561–579. DOI: [10.1007/978-3-030-99524-9\\_31](https://doi.org/10.1007/978-3-030-99524-9_31).
- [41] D. Beyer and S. Kanav. *Reproduction Package for STTT submission ‘Cooperative Verification: A literature review’*. Zenodo. 2023. DOI: [10.5281/zenodo.7838608](https://doi.org/10.5281/zenodo.7838608).
- [42] D. Beyer and S. Kanav. *Reproduction Package for TACAS 2022 Article ‘CoVeriTeam: On-Demand Composition of Cooperative Verification Systems’*. Zenodo. 2021. DOI: [10.5281/zenodo.5644953](https://doi.org/10.5281/zenodo.5644953).
- [43] D. Beyer, S. Kanav, and C. Richter. “Construction of Verifier Combinations Based on Off-the-Shelf Verifiers”. In: *Proc. FASE*. Springer, 2022, pp. 49–70. DOI: [10.1007/978-3-030-99429-7\\_3](https://doi.org/10.1007/978-3-030-99429-7_3).
- [44] D. Beyer, S. Kanav, and C. Richter. *Reproduction Package for FASE 2022 Article ‘Construction of Verifier Combinations Based on Off-the-Shelf Verifiers’*. Zenodo. 2022. DOI: [10.5281/zenodo.5812021](https://doi.org/10.5281/zenodo.5812021).
- [45] D. Beyer, S. Kanav, and H. Wachowitz. “CoVeriTeam Webservice: Verification as a Service”. In: *Proc. ICSE-Companion*. to appear. IEEE, 2023.
- [46] D. Beyer, S. Kanav, and H. Wachowitz. *Reproduction Package for Article ‘CoVeriTeam Service: Verification as a Service’*. Zenodo. 2022. DOI: [10.5281/zenodo.7276532](https://doi.org/10.5281/zenodo.7276532).
- [47] D. Beyer, S. Kanav, and H. Wachowitz. *Source-Code Repository of CoVeriTeam*. <https://gitlab.com/sosy-lab/software/coveriteam>. Accessed: 2023-02-09.
- [48] D. Beyer, S. Kanav, and H. Wachowitz. *Source-Code Repository of CoVeriTeam SERVICE*. <https://gitlab.com/sosy-lab/software/coveriteam-service>. Accessed: 2023-03-10.
- [49] D. Beyer, S. Kanav, T. Kleinert, and C. Richter. “Construction of Verifier Combinations From Off-the-Shelf Components”. In: Springer, 2025. DOI: [10.1007/s10703-024-00449-y](https://doi.org/10.1007/s10703-024-00449-y).
- [50] D. Beyer, S. Kanav, T. Kleinert, and C. Richter. *Reproduction Package for FMSD Article ‘Construction of Verifier Combinations From Off-the-Shelf Components’*. Zenodo. 2022. DOI: [10.5281/zenodo.7838348](https://doi.org/10.5281/zenodo.7838348).

- [51] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler. “Precision reuse for efficient regression verification”. In: *Proc. FSE*. ACM, 2013, pp. 389–399. DOI: [10.1145/2491411.2491429](https://doi.org/10.1145/2491411.2491429).
- [52] D. Beyer, S. Löwe, and P. Wendler. “Reliable Benchmarking: Requirements and Solutions”. In: *Int. J. Softw. Tools Technol. Transfer* 21.1 (2019), pp. 1–29. DOI: [10.1007/s10009-017-0469-y](https://doi.org/10.1007/s10009-017-0469-y).
- [53] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. “Conditional Model Checking: A Technique to Pass Information between Verifiers”. In: *Proc. FSE*. ACM, 2012. DOI: [10.1145/2393596.2393664](https://doi.org/10.1145/2393596.2393664).
- [54] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. “The Software Model Checker BLAST”. In: *Int. J. Softw. Tools Technol. Transfer* 9.5-6 (2007), pp. 505–525. DOI: [10.1007/s10009-007-0044-z](https://doi.org/10.1007/s10009-007-0044-z).
- [55] D. Beyer and T. Lemberger. “Conditional Testing: Off-the-Shelf Combination of Test-Case Generators”. In: *Proc. ATVA*. LNCS 11781. Springer, 2019, pp. 189–208. DOI: [10.1007/978-3-030-31784-3\\_11](https://doi.org/10.1007/978-3-030-31784-3_11).
- [56] D. Dietsch, M. Heizmann, A. Nutz, C. Schätzle, and F. Schüssele. “ULTIMATE TAIPAN with Symbolic Interpretation and Fluid Abstractions (Competition Contribution)”. In: *Proc. TACAS (2)*. LNCS 12079. Springer, 2020, pp. 418–422. DOI: [10.1007/978-3-030-45237-7\\_32](https://doi.org/10.1007/978-3-030-45237-7_32).
- [57] D. Kroening, M. Lewis, and G. Weissenbacher. “Proving Safety with Trace Automata and Bounded Model Checking”. In: *Proc. FM*. Springer, 2015, pp. 325–341. ISBN: 9783319192482, 9783319192499. DOI: [10.1007/978-3-319-19249-9\\_21](https://doi.org/10.1007/978-3-319-19249-9_21).
- [58] D. Kroening, M. Lewis, and G. Weissenbacher. “Under-Approximating Loops in C Programs for Fast Counterexample Detection”. In: *Proc. CAV*. Springer, 2013, pp. 381–396. ISBN: 9783642397981, 9783642397998. DOI: [10.1007/978-3-642-39799-8\\_26](https://doi.org/10.1007/978-3-642-39799-8_26).
- [59] D. Kröning and M. Tautschnig. “CBMC: C Bounded Model Checker (Competition Contribution)”. In: *Proc. TACAS*. LNCS 8413. Springer, 2014, pp. 389–391. DOI: [10.1007/978-3-642-54862-8\\_26](https://doi.org/10.1007/978-3-642-54862-8_26).
- [60] D. Neider, P. Garg, P. Madhusudan, S. Saha, and D. Park. “Invariant Synthesis for Incomplete Verification Engines”. In: *Proc. TACAS*. Springer, 2018, pp. 232–250. ISBN: 9783319899596, 9783319899602. DOI: [10.1007/978-3-319-89960-2\\_13](https://doi.org/10.1007/978-3-319-89960-2_13).
- [61] E. Chaudhary and S. Joshi. “PINAKA: Symbolic Execution meets Incremental Solving (Competition Contribution)”. In: *Proc. TACAS (3)*. LNCS 11429. Springer, 2019, pp. 234–238. DOI: [10.1007/978-3-030-17502-3\\_20](https://doi.org/10.1007/978-3-030-17502-3_20).
- [62] E. M. Clarke, D. Kröning, and F. Lerda. “A Tool for Checking ANSI-C Programs”. In: *Proc. TACAS*. LNCS 2988. Springer, 2004, pp. 168–176. DOI: [10.1007/978-3-540-24730-2\\_15](https://doi.org/10.1007/978-3-540-24730-2_15).

- [63] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. “Counterexample-Guided Abstraction Refinement”. In: *Proc. CAV*. LNCS 1855. Springer, 2000, pp. 154–169. DOI: [10.1007/10722167\\_15](https://doi.org/10.1007/10722167_15).
- [64] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. *Handbook of Model Checking*. Springer, 2018. ISBN: 978-3-319-10574-1. DOI: [10.1007/978-3-319-10575-8](https://doi.org/10.1007/978-3-319-10575-8).
- [65] F. He, Z. Sun, and H. Fan. “DEAGLE: An SMT-based Verifier for Multi-threaded Programs (Competition Contribution)”. In: *Proc. TACAS (2)*. LNCS 13244. Springer, 2022, pp. 424–428. DOI: [10.1007/978-3-030-99527-0\\_25](https://doi.org/10.1007/978-3-030-99527-0_25).
- [66] F. Lanzinger, A. Weigl, M. Ulbrich, and W. Dietl. “Scalability and precision by combining expressive type systems and deductive verification”. In: *PACMPL* 5.OOPSLA (Oct. 2021), pp. 1–29. ISSN: 2475-1421. DOI: [10.1145/3485520](https://doi.org/10.1145/3485520).
- [67] G. Vrettas and M. Sanderson. “Conferences versus journals in computer science”. In: *J. Assoc. Inf. Sci. Technol.* 66.12 (2015), pp. 2674–2684. DOI: [10.1002/asi.23349](https://doi.org/10.1002/asi.23349).
- [68] X. Ge, K. Taneja, T. Xie, and N. Tillmann. “DyTa: Dynamic Symbolic Execution Guided with Static Verification Results”. In: *Proc. ICSE*. ACM, 2011, pp. 992–994. DOI: [10.1145/1985793.1985971](https://doi.org/10.1145/1985793.1985971).
- [69] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. “Compositional May-must Program Analysis: Unleashing the Power of Alternation”. In: *Proc. POPL*. ACM, 2010, pp. 43–56. DOI: [10.1145/1706299.1706307](https://doi.org/10.1145/1706299.1706307).
- [70] H. Zhu, A. V. Nori, and S. Jagannathan. “Dependent Array Type Inference from Tests”. In: *Proc. VMCAI*. Springer, 2015, pp. 412–430. ISBN: 9783662460801, 9783662460818. DOI: [10.1007/978-3-662-46081-8\\_23](https://doi.org/10.1007/978-3-662-46081-8_23).
- [71] I. Dillig, T. Dillig, B. Li, and K. L. McMillan. “Inductive invariant generation via abductive inference”. In: *Proc. OOPSLA*. ACM, 2013, pp. 443–456. DOI: [10.1145/2509136.2509511](https://doi.org/10.1145/2509136.2509511).
- [72] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. “Software Verification for Weak Memory via Program Transformation”. In: *Proc. ESOP*. Springer, 2013, pp. 512–532. ISBN: 9783642370359, 9783642370366. DOI: [10.1007/978-3-642-37036-6\\_28](https://doi.org/10.1007/978-3-642-37036-6_28).
- [73] J. C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* 19.7 (1976), pp. 385–394. DOI: [10.1145/360248.360252](https://doi.org/10.1145/360248.360252).
- [74] J. Choi, J. Jang, C. Han, and S. K. Cha. “Grey-Box Concolic Testing on Binary Code”. In: *Proc. ICSE*. IEEE, May 2019. DOI: [10.1109/icse.2019.00082](https://doi.org/10.1109/icse.2019.00082).
- [75] J. H. Siddiqui and S. Khurshid. “Scaling symbolic execution using ranged analysis”. In: *Proc. SPLASH*. Ed. by Gary T. Leavens and Matthew B. Dwyer. ACM, 2012, pp. 523–536. DOI: [10.1145/2384616.2384654](https://doi.org/10.1145/2384616.2384654).

- [76] J. Haltermann and H. Wehrheim. “CoVEGI: Cooperative Verification via Externally Generated Invariants”. In: *Proc. FASE*. LNCS 12649. 2021, pp. 108–129. DOI: [10.1007/978-3-030-71500-7\\_6](https://doi.org/10.1007/978-3-030-71500-7_6).
- [77] J. Haltermann, M.-C. Jakobs, C. Richter, and H. Wehrheim. “Parallel Program Analysis via Range Splitting”. In: *Proc. FASE*. to appear. Springer, 2023.
- [78] J. R. Rice. “The Algorithm Selection Problem”. In: *Advances in Computers* 15 (1976), pp. 65–118. DOI: [10.1016/S0065-2458\(08\)60520-3](https://doi.org/10.1016/S0065-2458(08)60520-3).
- [79] J. Roemer, K. Genç, and M. D. Bond. “High-coverage, unbounded sound predictive race detection”. In: *Proc. PLDI*. ACM, June 2018. DOI: [10.1145/3192366.3192385](https://doi.org/10.1145/3192366.3192385).
- [80] J. Švejda, P. Berger, and J.-P. Katoen. “Interpretation-Based Violation Witness Validation for C: NITWIT”. In: *Proc. TACAS*. LNCS 12078. Springer, 2020, pp. 40–57. DOI: [10.1007/978-3-030-45190-5\\_3](https://doi.org/10.1007/978-3-030-45190-5_3).
- [81] J.-L. Lions, L. Lübeck, J.-L. Fauquembergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, and C. O’Halloran. *Ariane 501 Inquiry Board Report*. Tech. rep. July 1996.
- [82] K. L. McMillan. *Symbolic Model Checking*. Springer, 1993. ISBN: 978-1-4615-3190-6. DOI: [10.1007/978-1-4615-3190-6](https://doi.org/10.1007/978-1-4615-3190-6).
- [83] K. Li, C. Reichenbach, C. Csallner, and Y. Smaragdakis. “Residual investigation: predictive and precise bug detection”. In: *Proc. ISSTA*. ACM, 2012, pp. 298–308. DOI: [10.1145/2338965.2336789](https://doi.org/10.1145/2338965.2336789).
- [84] L. Holík, M. Kotoun, P. Peringer, V. Šoková, M. Trtík, and T. Vojnar. “PREDATOR Shape Analysis Tool Suite”. In: *Hardware and Software: Verification and Testing*. LNCS 10028. Springer, 2016, pp. 202–209. DOI: [10.1007/978-3-319-49052-6\\_13](https://doi.org/10.1007/978-3-319-49052-6_13).
- [85] L. M. de Moura and N. Bjørner. “Z3: An Efficient SMT Solver”. In: *Proc. TACAS*. LNCS 4963. Springer, 2008, pp. 337–340. DOI: [10.1007/978-3-540-78800-3\\_24](https://doi.org/10.1007/978-3-540-78800-3_24).
- [86] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler. “GRT: Program-Analysis-Guided Random Testing (T)”. In: *Proc. ASE*. IEEE, Nov. 2015. DOI: [10.1109/ase.2015.49](https://doi.org/10.1109/ase.2015.49).
- [87] C. Lattner and V. Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE. DOI: [10.1109/cgo.2004.1281665](https://doi.org/10.1109/cgo.2004.1281665).
- [88] M. Afzal, A. Asia, A. Chauhan, B. Chimdyalwar, P. Darke, A. Datar, S. Kumar, and R. Venkatesh. “VERIABS: Verification by Abstraction and Test Generation”. In: *Proc. ASE*. 2019, pp. 1138–1141. DOI: [10.1109/ASE.2019.00121](https://doi.org/10.1109/ASE.2019.00121).
- [89] M. Chalupa, A. Řečtáčková, V. Mihalkovič, L. Zaoral, and J. Strejček. “SYMBIOTIC 9: String Analysis and Backward Symbolic Execution with Loop Folding (Competition Contribution)”. In: *Proc. TACAS (2)*. LNCS 13244. Springer, 2022, pp. 462–467. DOI: [10.1007/978-3-030-99527-0\\_32](https://doi.org/10.1007/978-3-030-99527-0_32).

- [90] M. Christakis, P. Müller, and V. Wüstholtz. “Collaborative Verification and Testing with Explicit Assumptions”. In: *Proc. FM*. LNCS 7436. Springer, 2012, pp. 132–146. DOI: [10.1007/978-3-642-32759-9\\_13](https://doi.org/10.1007/978-3-642-32759-9_13).
- [91] M. Czech, Eyke Hüllermeier, Marie-Christine Jakobs, and Heike Wehrheim. “Predicting rankings of software verification tools”. In: *Proc. SWAN*. ACM, 2017, pp. 23–26. DOI: [10.1145/3121257.3121262](https://doi.org/10.1145/3121257.3121262).
- [92] M. Dangl, S. Löwe, and P. Wendler. “CPACHECKER with Support for Recursive Programs and Floating-Point Arithmetic (Competition Contribution)”. In: *Proc. TACAS*. LNCS 9035. Springer, 2015, pp. 423–425. DOI: [10.1007/978-3-662-46681-0\\_34](https://doi.org/10.1007/978-3-662-46681-0_34).
- [93] M. Giacobbe, D. Kroening, and J. Parsert. “Neural termination analysis”. In: *Proc. ESEC/FSE*. ACM, 2022, pp. 633–645. DOI: [10.1145/3540250.3549120](https://doi.org/10.1145/3540250.3549120).
- [94] M. Heizmann, Y.-F. Chen, D. Dietsch, M. Greitschus, J. Hoenicke, Y. Li, A. Nutz, B. Musa, C. Schilling, T. Schindler, and A. Podelski. “ULTIMATE AUTOMIZER and the Search for Perfect Interpolants (Competition Contribution)”. In: *Proc. TACAS (2)*. LNCS 10806. Springer, 2018, pp. 447–451. DOI: [10.1007/978-3-319-89963-3\\_30](https://doi.org/10.1007/978-3-319-89963-3_30).
- [95] M. Kotoun, P. Perring, V. Šoková, and T. Vojnar. “Optimized PREDATORHP and the SV-COMP Heap and Memory Safety Benchmark (Competition Contribution)”. In: *Proc. TACAS*. LNCS 9636. Springer, 2016, pp. 942–945. DOI: [10.1007/978-3-662-49674-9\\_66](https://doi.org/10.1007/978-3-662-49674-9_66).
- [96] M. Li, Y. Chen, L. Wang, and G. Xu. “Dynamically validating static memory leak warnings”. In: *Proc. ISSSTA*. ACM, July 2013. DOI: [10.1145/2483760.2483778](https://doi.org/10.1145/2483760.2483778).
- [97] M. V. Hermenegildo. “Conferences vs. Journals in CS, what to do? Evolutionary ways forward and the ICLP/TPLP model”. In: *Dagstuhl 12452: Publication Culture in Computing Research - Position Papers*. Vol. 12452. 2012.
- [98] M. Y. R. Gadelha, F. R. Monteiro, L. C. Cordeiro, and D. A. Nicole. “ESBMC v6.0: Verifying C Programs using  $k$ -Induction and Invariant Inference (Competition Contribution)”. In: *Proc. TACAS (3)*. LNCS 11429. Springer, 2019, pp. 209–213. DOI: [10.1007/978-3-030-17502-3\\_15](https://doi.org/10.1007/978-3-030-17502-3_15).
- [99] M. Y. Vardi. “Conferences vs. Journals in Computing Research”. In: *Commun. ACM* 52.5 (May 2009), p. 5. DOI: [10.1145/1506409.1506410](https://doi.org/10.1145/1506409.1506410).
- [100] N. G. Leveson and C. S. Turner. “An investigation of the Therac-25 accidents”. In: *Computer* 26.7 (1993), pp. 18–41. DOI: [10.1109/MC.1993.274940](https://doi.org/10.1109/MC.1993.274940).
- [101] 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. Springer, 2014, pp. 585–602. DOI: [10.1007/978-3-319-08867-9\\_39](https://doi.org/10.1007/978-3-319-08867-9_39).

- [102] P. Ayaziová, M. Chalupa, and J. Strejček. “SYMBIOTIC-WITCH: A Klee-Based Violation Witness Checker (Competition Contribution)”. In: *Proc. TACAS (2)*. LNCS 13244. Springer, 2022, pp. 468–473. DOI: [10.1007/978-3-030-99527-0\\_33](https://doi.org/10.1007/978-3-030-99527-0_33).
- [103] P. Garg, D. Neider, P. Madhusudan, and D. Roth. “Learning invariants using decision trees and implication counterexamples”. In: *Proc. POPL*. ACM, Jan. 2016. DOI: [10.1145/2837614.2837664](https://doi.org/10.1145/2837614.2837664).
- [104] P. Müller, P. Peringer, and T. Vojnar. “PREDATOR Hunting Party (Competition Contribution)”. In: *Proc. TACAS*. LNCS 9035. Springer, 2015, pp. 443–446. DOI: [10.1007/978-3-662-46681-0\\_40](https://doi.org/10.1007/978-3-662-46681-0_40).
- [105] P. Wendler. “CPACHECKER with Sequential Combination of Explicit-State Analysis and Predicate Analysis (Competition Contribution)”. In: *Proc. TACAS*. LNCS 7795. Springer, 2013, pp. 613–615. DOI: [10.1007/978-3-642-36742-7\\_45](https://doi.org/10.1007/978-3-642-36742-7_45).
- [106] R. Baldoni, E. Coppa, D. C. D’Elia, C. Demetrescu, and I. Finocchi. “A Survey of Symbolic Execution Techniques”. In: *ACM Comput. Surv.* 51.3 (2018), 50:1–50:39. DOI: [10.1145/3182657](https://doi.org/10.1145/3182657).
- [107] R. Jhala and R. Majumdar. “Software Model Checking”. In: *ACM Computing Surveys* 41.4 (2009). DOI: [10.1145/1592434.1592438](https://doi.org/10.1145/1592434.1592438).
- [108] R. Xu, J. Chen, and F. He. “Data-driven loop bound learning for termination analysis”. In: *Proc. ICSE*. ACM, May 2022. DOI: [10.1145/3510003.3510220](https://doi.org/10.1145/3510003.3510220).
- [109] C. Richter and H. Wehrheim. “Attend and represent: a novel view on algorithm selection for software verification”. In: *Proc. ASE*. 2020, pp. 1016–1028. DOI: [10.1145/3324884.3416633](https://doi.org/10.1145/3324884.3416633).
- [110] S. Arlt, A. Podelski, and M. Wehrle. “Reducing GUI test suites via program slicing”. In: *Proc. ISSTA*. ACM, 2014. DOI: [10.1145/2610384.2610391](https://doi.org/10.1145/2610384.2610391).
- [111] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. “Differential assertion checking”. In: *Proc. FSE*. ACM, 2013, pp. 345–355. DOI: [10.1145/2491411.2491452](https://doi.org/10.1145/2491411.2491452).
- [112] S. Kumar, A. Sanyal, R. Venkatesh, and P. Shah. “Property Checking Array Programs Using Loop Shrinking”. In: *Proc. TACAS*. Springer, 2018, pp. 213–231. ISBN: 9783319899596, 9783319899602. DOI: [10.1007/978-3-319-89960-2\\_12](https://doi.org/10.1007/978-3-319-89960-2_12).
- [113] S. Löwe, M. U. Mandrykin, and P. Wendler. “CPACHECKER with Sequential Combination of Explicit-Value Analyses and Predicate Analyses (Competition Contribution)”. In: *Proc. TACAS*. LNCS 8413. Springer, 2014, pp. 392–394. DOI: [10.1007/978-3-642-54862-8\\_27](https://doi.org/10.1007/978-3-642-54862-8_27).
- [114] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. “Automatic Predicate Abstraction of C Programs”. In: *Proc. PLDI*. ACM, 2001, pp. 203–213. DOI: [10.1145/378795.378846](https://doi.org/10.1145/378795.378846).



- [115] T. Ball and S. K. Rajamani. “The SLAM project: Debugging System Software via Static Analysis”. In: *Proc. POPL*. ACM, 2002, pp. 1–3. DOI: [10.1145/503272.503274](https://doi.org/10.1145/503272.503274).
- [116] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002. DOI: [10.1007/3-540-45949-9](https://doi.org/10.1007/3-540-45949-9).
- [117] M. Polo Usaola, P. Reales Mateo, and B. Pérez Lamancha. “Reduction of Test Suites Using Mutation”. In: *Proc. FASE*. Springer, 2012, pp. 425–438. ISBN: 9783642288715, 9783642288722. DOI: [10.1007/978-3-642-28872-2\\_29](https://doi.org/10.1007/978-3-642-28872-2_29).
- [118] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. “The Art, Science, and Engineering of Fuzzing: A Survey”. In: *IEEE Trans. Software Eng.* 47.11 (2021), pp. 2312–2331. DOI: [10.1109/TSE.2019.2946563](https://doi.org/10.1109/TSE.2019.2946563).
- [119] V. Malík, P. Schrammel, and T. Vojnar. “2LS: Heap Analysis and Memory Safety (Competition Contribution)”. In: *Proc. TACAS (2)*. LNCS 12079. Springer, 2020, pp. 368–372. DOI: [10.1007/978-3-030-45237-7\\_22](https://doi.org/10.1007/978-3-030-45237-7_22).
- [120] V. Wüstholtz and M. Christakis. “Targeted greybox fuzzing with static lookahead analysis”. In: *Proc. ICSE*. ACM, June 2020. DOI: [10.1145/3377811.3380388](https://doi.org/10.1145/3377811.3380388).
- [121] W. Leeson and M. Gerrard. *Graves-Par: Selecting a Verification Portfolio Using Efficient Graph Neural Networks*. <https://github.com/mgerrard/graves-par>. 2023.
- [122] Y. Cai, C. Ye, Q. Shi, and C. Zhang. “Peahen: fast and precise static deadlock detection via context reduction”. In: *Proc. ESEC/FSE*. ACM, 2022, pp. 784–796. DOI: [10.1145/3540250.3549110](https://doi.org/10.1145/3540250.3549110).
- [123] Y. Demyanova, T. Pani, H. Veith, and F. Zuleger. “Empirical software metrics for benchmarking of verification tools”. In: *Formal Methods in System Design* 50.2-3 (2017), pp. 289–316. DOI: [10.1007/s10703-016-0264-5](https://doi.org/10.1007/s10703-016-0264-5).
- [124] Y. Koroglu and A. Sen. “TCM: Test Case Mutation to Improve Crash Detection in Android”. In: *Proc. FASE*. Springer, 2018, pp. 264–280. ISBN: 9783319893624, 9783319893631. DOI: [10.1007/978-3-319-89363-1\\_15](https://doi.org/10.1007/978-3-319-89363-1_15).
- [125] Y. Noller, R. Kersten, and C. S. Pasareanu. “Badger: Complexity analysis with fuzzing and symbolic execution”. In: *Proc. ISSSTA*. ACM, 2018, pp. 322–332. DOI: [10.1145/3213846.3213868](https://doi.org/10.1145/3213846.3213868).
- [126] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. “The Matter of Heartbleed”. In: *Proc. IMC*. ACM, 2014, pp. 475–488. DOI: [10.1145/2663716.2663755](https://doi.org/10.1145/2663716.2663755).

# A Credits

## CoVeriTeam: On-Demand Composition of Cooperative Verification Systems

The article in [Section B.2](#) is authored by Dirk Beyer and Sudeep Kanav and published by Springer in the proceedings of TACAS 2022 [40]. A [reproduction package](#) [42] is available and the tool CoVeriTeam is [available open source](#).

Sudeep Kanav is the main author of this article and contributed about 80 % to the content of this article.

## Construction of Verifier Combinations From Off-the-Shelf Components

The article in [Section B.3](#) is authored by Dirk Beyer, Sudeep Kanav, Tobias Kleinert, and Cedric Richter and is under review for publication in the *FASE 2022 special issue* in the Springer journal [Formal Methods in System Design](#) [49]. A [reproduction package](#) [50] is available online.

This is an extended version of the article *Construction of Verifier Combinations Based on Off-the-Shelf Verifiers* authored by Dirk Beyer, Sudeep Kanav, and Cedric Richter and published by Springer in the proceedings of FASE 2022 [43]. A [reproduction package](#) [44] is available online.

Sudeep Kanav is the main author of both of these articles and contributed about 70 % to the content of each of them.

## Cooperative Software Verification: An Exploratory Literature Review

The article in [Section B.1](#) is authored by Dirk Beyer and Sudeep Kanav. It is under review for publication in the Springer journal [International Journal on Software Tools for Technology Transfer](#) [39]. A [reproduction package](#) [41] is available online.

Sudeep Kanav is the main author of this article and contributed about 80 % to the content of this article.

## CoVeriTeam Webservice: Verification as a Service

The article in [Section B.4](#) is authored by Dirk Beyer, Sudeep Kanav, and Henrik Wachowitz. It is in process for publication in the companion proceedings of ICSE 2023 by IEEE [45]. A [reproduction package](#) [46] is available online and the



source code of the service is [available open source](#). The web-service is available at: <https://coveriteam-service.sosy-lab.org/>



Sudeep Kanav and Henrik Wachowitz are the main authors and contributed equally to this article.

## B Original Manuscripts

This appendix includes all publications discussed in [Chapters 3, 4, 5, and 6](#). Publications are ordered according to their appearance in this work. In these publications, authors are listed in alphabetical order.

Noname manuscript No.  
 (will be inserted by the editor)

# Cooperative Software Verification: An Exploratory Literature Review

Dirk Beyer  and Sudeep Kanav 

the date of receipt and acceptance should be inserted later

**Abstract** Cooperative verification comprises verification approaches in which multiple verification components are employed, and the employed verification components exchange information to solve a verification task together. In addition to exploiting synergies between different verification approaches, cooperative verification focuses on the interaction between components, highlighting their roles and the exchanged information.

The literature on software verification has not been reviewed from this perspective. We aim to fill this gap by providing an overview of cooperative verification techniques. We have systematically reviewed the scientific literature to identify combination techniques and classify them based on the role of the exchanged information, with an emphasis on cooperative approaches. We focus on automated-reasoning techniques that analyze source code.

**Keywords** Cooperative Verification · Software Verification · Automatic Verification · Collaborative Verification

## 1 Introduction

The verification of computer programs is an important problem, because the correct functioning of our society and economy depends on the correct functioning of computer programs that are ubiquitous in our lives, including transportation, energy, and financial systems. However, the verification of computer programs is an undecidable problem [228]. As a result, we cannot expect a general solution: we need to use heuristic approaches and exploit available techniques.

In the past decades, numerous algorithms, tools, and techniques have been developed to solve specific classes of the software-verification problem (e.g., [31, 53, 67, 68, 74, 125, 144, 190]). Each of these tools and techniques is strong in some areas and limited in others, which makes it imperative to consider combinations to address the problem. According to a recently proposed schema [38], these combinations can be classified into integrated combinations (e.g., [14, 23, 78, 106, 118]) and combinations of off-the-shelf components. The latter can be further classified into portfolios [133, 148, 238], algorithm selection [36, 203, 204], and cooperative approaches. The key idea in a cooperative combination is to make use of the capabilities of a tool or a technique to complement the capabilities of another. In a cooperative combination, participating actors exchange information, helping each other to solve the given verification task. Examples of cooperative techniques are conditional model checking [24], conditional testing [33], and cooperative verification via externally generated invariants [123]. In contrast, in a non-cooperative combination, participating tools do not exchange information.

The primary difference between a cooperative and a non-cooperative combination is the use of information produced by one participant in the combination by other participants in the combination. This information is exchanged through the means of *artifacts*. Identifying the information that can be used for cooperation and exporting it in an exchangeable format enables the reuse of analysis tools as components. Ideally, one should be readily able to replace any component in a cooperative combination by other components with the same interface. This survey aims to provide an overview of such approaches in the published literature.

Cooperation between verification tools and techniques has the potential to improve the performance

and effectiveness of software verification. During our research in the past few years, we have found that cooperative verification has been addressed by a number of researchers. Several combination approaches focusing on off-the-shelf combinations and information exchange have been published in the scientific literature in the last decade [22, 24, 64, 65, 95, 123]. These articles may use different terms (e.g. collaborative verification, tool combination, etc.) instead of *cooperative verification*, but they are essentially combination approaches that are based on information exchange.

Although there have been various literature reviews for software verification [12, 35, 138, 141, 142, 173], a thorough literature review on cooperative verification had yet to be conducted. We aim to address this gap with our literature review. Our goal is to provide an overview of cooperative verification techniques published in the scientific literature and organize this knowledge by developing a classification of these techniques.

We systematically searched the articles published in leading scientific publications related to the area of software verification. We followed a multi-stage selection process. First, we filtered the articles based on keyword searches on the titles and abstracts. Then, we pruned our selection by manually processing the titles and abstracts of these articles. At last, we analyzed the relevant articles to decide if they present a cooperative technique.

Following the above-mentioned multi-stage search process, we identified 72 articles that present a cooperative verification technique, according to our definition. We then developed a classification for these selected techniques based on the role that the exchanged information has in the verification process. Our survey focuses on automated reasoning techniques that analyze source code.

We make the following contributions:

1. a definition of cooperative verification, making the abstract concept more concrete,
2. an overview of cooperative verification techniques published in the scientific literature during the period of 2012–2022,
3. a classification of cooperative verification techniques based on the role of the exchanged information, and
4. an artifact containing the data used and produced by the stages of our selection methodology.

## 2 Related Work

We have divided the related work into three parts: (1) literature published before the considered time frame, (2) non-cooperative combinations, (3) literature reviews

related to software verification, and (4) frameworks for tool combinations.

### 2.1 Literature Published Before the Considered Time Frame

The research on software model checking became industrially relevant in the first decade of this century [35]. As standalone techniques started to mature, combinations of techniques began to emerge, starting around 2005. Such combinations include using dynamic analysis to validate the results of a static checker [78, 79, 106], an approach by Majumdar and Sen [172] that combines random testing and concolic testing [112, 211], SYNERGY [118] combining testing and verification, DASH [14] building on and advancing SYNERGY by using the information from failed tests to refine the abstraction, an approach combining abstract interpretation and bounded model checking [197], and SMASH [113] combining predicate abstraction [116] and testing. Furthermore, several approaches mentioned in Section 2.4 were published before the considered time frame.

### 2.2 Memoization and (De-) Composition

We are investigating those combinations in which more than one component exchange information in form of artifacts. We differentiate cooperation from decomposition and memoization approaches, both of which are used to improve performance.

Memoization approaches attempt to improve scalability in inter-procedural analysis by saving and reusing an over-approximation of a block of code called function (or procedure) summaries [7, 19, 20, 111, 130, 212, 213, 240]. In most of the cases, it is the same analysis that is saving information for future use. We have excluded most of these approaches due to this reason. We found one case in which such summaries were used by some other analysis and have included it in our review [160].

Some verification approaches structurally decompose the given program into several smaller instances, and then analyze these smaller instances using an off-the-shelf verifier [59, 189, 216]. Also, some works propose to run several instance of an analysis in parallel and to share information (e.g., clauses, lemmas) among these parallel instances [189, 243]. The shared information is implementation-specific. Intuitively, all the above ideas can be called *self-cooperation*, whereas this survey is concerned with cooperation among different components.

Assume-Guarantee reasoning [9, 139, 196] is a well known approach where the proof obligations for the complete system are decomposed into proof obligations

for smaller components. These specifications are often called contracts and expressed in the form of assumptions and guarantees, i.e., the behavior of a given component satisfies the guarantees if it is executed in an environment that satisfies the assumptions. Contracts for atomic components are verified against the implementation and these contracts are then hierarchically composed and checked, resulting in the verification result for the complete system. These contracts are written and decomposed manually, so these approaches in general do not fit in our definition.

There is a line of research proposing to automate the process by inferring or learning the assumptions (and guarantees) of components [10, 57, 60, 70, 90, 107]. However, many of these articles were published before the time-frame we considered, and others were filtered out by our process.

*Integrated Approaches.* Since the invention of abstract interpretation [74], various attempts have been made to combine different abstract domains [71, 75, 119, 162]. These techniques attempt to use information from multiple abstract domains to solve a verification task and aim to improve the efficiency of the analysis.

Later, combinations of data-flow analysis and model checking were developed [67, 97, 127]. The key idea in these approaches was to traverse the state space in the abstract domain for efficiency and use the concrete states to increase the precision for dealing with spurious counterexamples. Then the above-discussed types of compositions —abstract domains and analysis techniques (data flow and model checking)— were unified [21, 25, 26].

These are typically well defined mathematical compositions guaranteeing soundness of the composition. However, the components in these compositions are tightly coupled and typically share information through implementation dependent data structures. The benefit of exporting all the information exchanged through the means of artifacts is unclear. We have excluded these type of compositions. We are interested in combinations that are loosely coupled and exchange information through standardized interfaces.

### 2.3 Literature Reviews Related to Software Verification

Numerous literature reviews on different aspects of software verification have been conducted. However, we review the scientific literature from a novel perspective of cooperative verification. Jhala and Majumdar [138] review the techniques for analyzing software systems. Kaleeswaran, Nordmann, Vogel, and Grunske [141] provide an overview of the state of the art for counterexample explanation. Karna, Chen, Yu, Zhong, and Zhao [142]

review the role of model checking in software engineering. The overview of the development of software model checking in the last two decades [35] includes a description of the history of software model checking and gives details about the raising attention that the topic attracted, in terms of published literature and in terms of developed tools for software verification. Fraser, Wotawa, and Ammann [100] survey the use of model checkers for generating tests. Baldoni, Coppa, D’Elia, Demetrescu, and Finocchi [12] survey symbolic-execution [144] techniques. D’Silva, Kröning, and Weissenbacher [89] review scientific literature for automated static-analysis techniques, focusing on abstract interpretation, model checking, and bounded model checking. Manès, Han, Han, Cha, Egele, Schwartz, and Woo [173] present a unified model of fuzzing and a taxonomy of the literature on fuzzing.

Beyer and Lee [32] survey the transformations used in modular verification approaches and advocate use of transformations and standard exchange formats for development of verification tools.

### 2.4 Frameworks for Tool Combinations

Computer scientists have discussed the advantages and disadvantages of developing modular systems by reusing components [109, 157, 215, 221, 222]. The verification community has also developed various frameworks and web services for creating tool combinations and orchestrating their execution.

The *Electronic Tools Integration* platform (ETI) [50, 175, 178, 223] was designed with the vision to allow users to access tools over the internet without needing to install them on the local machine. A user can also execute these tools in combinations. The information is exchanged between the tools with the help of taxonomic specifications and LTL synthesis. jETI [176, 177], a redesign of ETI platform, reduces the effort for integrating and updating tools. It uses web services to provide lightweight remote integration and coordination of verification tools.

The *Evidential Tool Bus* (ETB) [76, 77, 208] uses a variant of Datalog [4, 56] for integration of tools. Its primary use case is building and maintaining assurance cases for the purpose of certification. It maintains a store of proven claims for files and their versions, which can later be reused as partial results in regression verification.

COVERTEAM [29] supports the creation and execution of tool combinations. It is based on the notions of actors and artifacts. Actors are interfaces of verification tools and artifacts are interfaces of the information

objects exchanged between the actors. An actor based on a concrete tool is defined with the help of configuration information like its type (type of its inputs and outputs), the location from where to download the tool, and tool-info module (a Python module used to prepare the command and parse the tool output), and resource limits to enforce. With the help of this information, CoVeriTEAM uses BENCHEXEC [34] to execute the tools in isolation, enforcing the resource limitations. A user can also create combinations of various actors in CoVeriTEAM.

SEAHORN [122] is an LLVM-based verification framework for checking safety properties of programs. It first converts a program to LLVM bitcode, generates verification condition as constrained Horn clauses (CHC), and then uses various off-the-shelf verifiers to discharge verification conditions.

### 3 Methodology

In this section, we define cooperative verification, our research goals, and the process for the literature selection.

#### 3.1 Definition of Cooperative Verification

We are interested in tools and approaches for analyzing source code that are themselves are combinations of other tools and approaches.

Figure 1 shows a layering of tools and libraries used in analysis. On top there are *analyzers* that reason about behavior of a system. Example of a system are: source code, an android app, a probabilistic model, or a theory. The analyzers typically encode the problem in a formalism such as BDD or SMT formula, or constraints and delegate the solving to *solvers*. An analyzer could either directly call the solver, or use an interface to the solver.

One can see the interaction between these layers as cooperation (e.g., an analyzer and a solver). There could also be cooperation among solvers. However, in this work, we focus explicitly on the cooperation between *analyzers*. With this in mind, we introduce a definition of cooperative verification.

Cooperative verification is an approach to verification in which multiple tools work together to solve a *verification task* by exchanging information. *Verification actors* produce and consume this information, and *verification artifacts* encapsulate and represent the exchanged information. In the following, we list a few examples for each of the notions defined above:

- verification actor: off-the-shelf tool, agent, web service, executable
- verification artifact: program, alarm, invariant, test
- verification problem: safety check, termination check, test generation, feasibility check, refinement
- analyzer: verifier, validator, test generator

We define cooperative verification as follows:

**Definition 1** A verification approach is called **cooperative**, if

1. *identifiable verification actors* pass information in form of
2. *identifiable verification artifacts* towards the common objective of
3. solving a verification problem,
4. where at least two of these actors are analyzers.

We impose the restriction of *at least two analyzers* to focus on the combinations comprising of components where each component can independently analyze (a part of) the problem. This restriction automatically excludes preprocessing, translations, and structural decomposition.

Cooperation is about making use of different tools and techniques. This automatically excludes standalone approaches (e.g., [250]), extensions to an existing approach (e.g., [45, 120]), tightly coupled combinations (which we call integrated combinations) (e.g., [31, 41, 126, 168]), and memoization mechanisms (e.g., [19, 232]).

#### 3.2 Research Goals

Our research objective is to review cooperative verification techniques published in the scientific literature, provide an overview, and systematically arrange the knowledge by classifying these techniques. We aim to address two concrete research goals:

- RG 1. Identify the cooperative verification techniques from the chosen sample of the published scientific literature.
- RG 2. Develop a classification for the identified cooperative verification techniques based on the role played by the exchanged information.

#### 3.3 Selection of Literature for Review

##### 3.3.1 Stage 0: Publication Selection

Cooperative verification is related to (1) formal methods (formal verification), (2) software engineering, and (3) semantics of programming languages. Table 1 lists

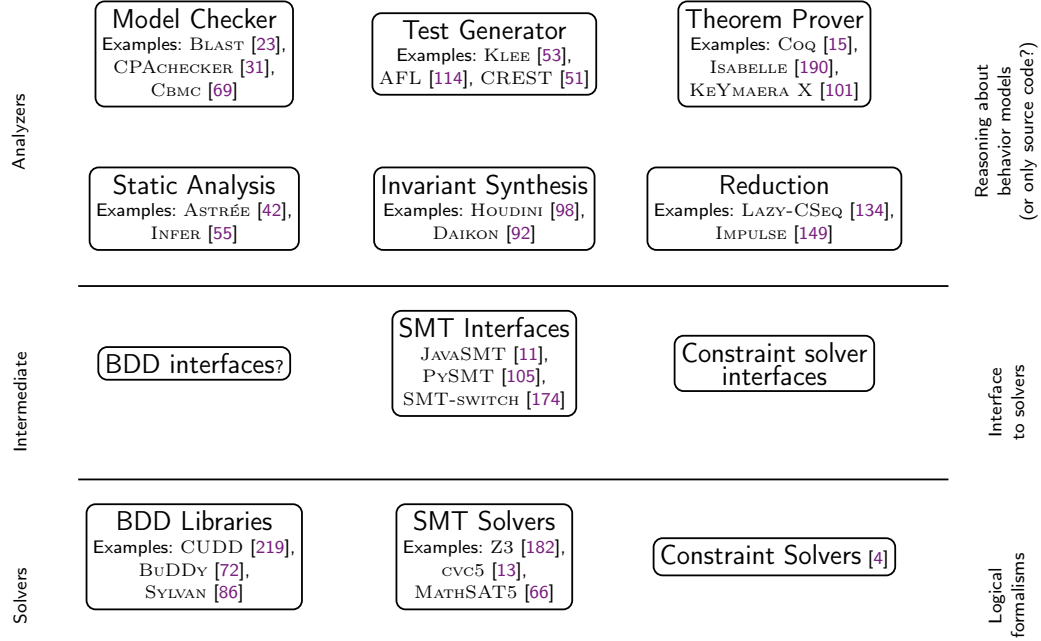


Fig. 1: Intuition for a cooperative approach

Table 1: Shortlisted publication outlets (conference proceedings) for the review

Abbreviation	Name	Publisher
<b>Conferences in Software Engineering</b>		
ASE	International Conference on Automated Software Engineering	ACM/IEEE
ESEC/FSE	Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering	ACM
ICSE	International Conference on Software Engineering	ACM/IEEE
ISSTA	International Symposium on Software Testing and Analysis	ACM
FASE	Fundamental Approaches to Software Engineering	Springer
SEFM	International Conference on Software Engineering and Formal Methods	Springer
<b>Conferences in Formal Methods</b>		
CAV	Computer Aided Verification	Springer
TACAS	Tools and Algorithms for the Construction and Analysis of Systems	Springer
ATVA	Automated Technology for Verification and Analysis	Springer
VMCAI	Verification, Model Checking, and Abstract Interpretation	Springer
FM	International Symposium on Formal Methods	Springer
<b>Conferences in Programming Languages</b>		
PLDI	Programming Language Design and Implementation	ACM
POPL	Principles of Programming Languages	ACM
OOPSLA	Object-Oriented Programming, Systems, Languages, and Applications	ACM
ESOP	European Symposium on Programming	Springer

the shortlisted publications. We considered the proceedings of the conferences in these fields that are ranked A

or A\* according to the CORE rankings [2] and published by ACM or Springer.<sup>1</sup>

We added four conferences to this list which are not A or A\* according to CORE rankings: ATVA, FASE, VMCAI, and SEFM.

Conferences are considered the first choice in computer science for publishing new results [129, 231, 234]. Therefore, we considered only conference proceedings for our survey. This means that journals were excluded from our consideration.

### 3.3.2 Stage 1: Study Selection

We considered the articles that satisfied all the following three conditions:

1. The article was published in the proceedings of the conferences listed in Table 1.
2. The year of publication is  $\geq 2012$ .
3. The length of the article is more than 7 pages for ACM/IEEE publications (i.e., two columns); and more than 10 pages for Springer (i.e., single column).

This gave us a corpus of 8 108 articles in total that were published in the last 10 years and were not short papers or extended abstracts. These papers comprised our *search universe*.

#### Justifications for the used heuristics.

We focused on research papers and long tool papers as these articles are more likely to contain a detailed description of a new technique. As a heuristic, we settled on more than 10 pages for one-column articles and more than 7 pages for two-column articles.

We considered last ten years (from the time when we started working on this survey) because we wanted to focus on the recent literature.

### 3.3.3 Stage 2: Search Query

We searched the abstracts and titles of the selected articles for the terms listed in Listing 1. This reduced the the number of articles to 4 332. This search was automated using a Python script.

The term *cooperative verification* was invented by Beyer and Wehrheim [37, 38] in 2019. When we search for this term in the complete text of papers in our search space, only a few pass. Therefore, we broadened our search by keeping the search terms more general (Listing 1). However, the terms *testing* and *verification* are so broad that the filter is ineffective.

<sup>1</sup> Both ASE and ICSE alternate with ACM and IEEE as publishers. This includes the important conferences relevant to this survey.

We experimented with our search query on a small subset of the articles. It showed that an article discussing a verification technique is highly likely to have at least one of these keywords in the title or abstract.

Listing 1 shows the search terms. Our search was conducted case-insensitive and using stems instead of complete words. This allows us to include different forms of a given word, e.g., search for *verif* would include *verify*, *verification*, *verified*, *verifier*, etc.

```
cooper OR collabor OR verif OR test OR
model check OR theorem prov OR program
analy OR data flow analy OR data-flow
analy OR static analy
```

Listing 1: Search query

### 3.3.4 Stage 3: Exclusion based on title

During this stage, we manually processed the titles of the papers. We pruned our search space to eliminate the studies that we were confident were not about cooperative verification. As mentioned before, our search query is too permissive; this results in many titles and abstracts passing the filter even if these articles do not present a verification technique. We kept the articles in consideration about which we could not confidently make this decision. The stage reduced the number of articles in consideration to 1 117.

### 3.3.5 Stage 4: Exclusion based on abstract

We analyzed the abstracts of the remaining papers and excluded the ones not discussing verification approaches employing two or more techniques. This further reduced the number of articles in consideration to 381.

### 3.3.6 Stage 5: Review and Classification

We reviewed the 381 articles that passed the filter from the previous stage. We applied our definition of cooperative verification to the techniques proposed in these articles. As a result of this process, we found that 72 articles present a cooperative verification technique.

*Review Process.* First, we weed out the papers not in scope e.g., not dealing with source code [?, 233], not fully automated approaches [214], etc. Then we decide if it is a combination of the kind we are interested in. This excludes standalone approaches (e.g., [250]), extensions to an approach (e.g., [183]), memoization techniques (e.g., [19, 232]), tightly coupled combinations (e.g., [61, 126]). We achieve this by trying to identify the actors and artifacts in the approach presented in a paper, identifying the information represented by the artifact,



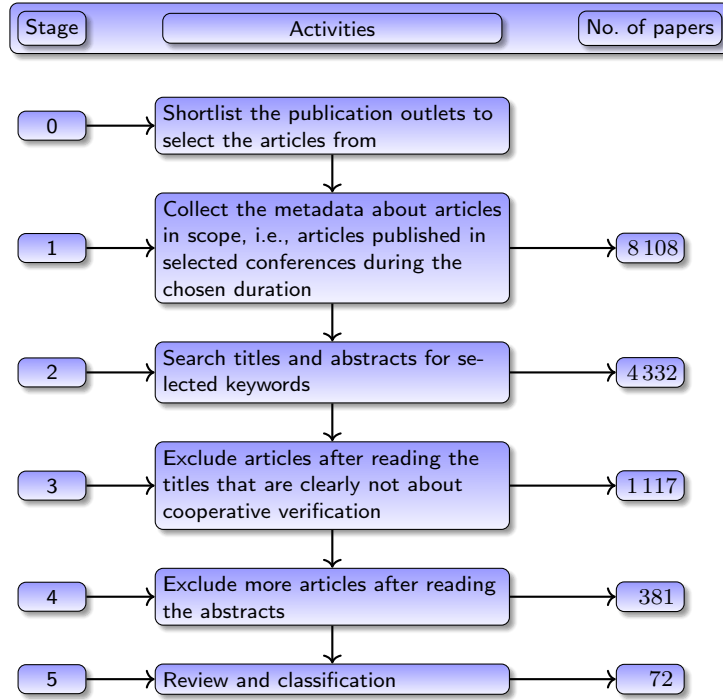


Fig. 2: Stages of the selection process and number of papers selected in each stage

Table 2: Classification of the cooperative verification techniques

Class name	Explanation	Examples	Count
Reduction	The verification task is reduced such that it can be solved by another analyzer.	[134, 150, 155]	12
Guide	The artifact produced by an analyzer acts to guide another analyzer.		
Conditional	First analyzer tries to solve the verification problem and produces an artifact that summarizes the work done; another actor then uses this information to focus only on the unsolved parts of the task.	[24, 33, 65]	12
Hint	An analyzer generates hints that are then used to guide the verification of another analyzer.	[62, 171, 241]	17
Scrutiny	The artifact produced is scrutinized by another analyzer.		
Validation	The result produced by one analyzer is validated by another analyzer.	[16, 17]	14
Refinement	The artifact produced by one analyzer is refined by another analyzer.	[5, 165]	3
Iterative Validation Guided Refinement	The artifact produced is first validated, and then the result of validation is used to guide the process of refinement. This sequence is repeated until a solution is found.	[22, 185]	14

inferring its relation to the input verification problem, and its role for the actor that consumes it. At last, we group them together resulting in our classification.

Ideally, all of these would have been easily identifiable with standardized exchanged format. However, that was not the case. If we restrict ourselves only to the papers that clearly state and import/export the artifacts in a standardized format then only a handful of papers would qualify. We were liberal in our interpretation and also included the papers where the artifact can be exported in an exchangeable format with some engineering effort.

*Exclusions.* We focus on the combination of analyzers that analyze source code. This excludes analysis of hardware, probabilistic models, binaries, and hybrid systems. Moreover, we focus on automated approaches, which excludes interactive methods.

Testing is a broad area covering different types of systems to many methods for generating test inputs. We include symbolic execution and generating tests using a model checker in consideration for analyzers. This results in exclusion of random testing, fuzzing, model-based testing, and evolutionary approaches for testing.

In the next two sections, we present the results of our survey. Section 4 presents the classification of the 72 articles describing cooperative verification techniques. During our review, we also found articles that were not cooperative techniques in themselves but presented a tool or a technique that could be used as a cooperative actor. Section 5 reports these cooperative actors.

#### 4 Classification of Selected Techniques

We classify the cooperative verification techniques based on the role of the exchanged information, i.e., the role of the artifact that is shared between the actors. Table 2 summarizes these classes, providing a brief description and the number of articles in each class. Figure 3 shows the class diagram for these classes.

##### 4.1 Reduction: Verification Task is Reduced

In this class of cooperative techniques, the given verification task is reduced by an analyzer such that an off-the-shelf analyzer can be used for its verification.

*Concurrent systems.* Various techniques targeting the verification of concurrent systems reduce the given concurrent system to a sequential one such that their behavior is the same concerning the property in question. An off-the-shelf verifier for sequential programs can then be used for the verification of the reduced program.

LAZY-CSEQ [134] reduces a multi-threaded C program into a non-deterministic sequential program and then employs an off-the-shelf bounded model checker for its verification. Alglave, Kröning, Nimal, and Tautschnig [6] present a combination that reduces the given program such that the verifiers assuming sequential consistency can be used to verify the program for weak memory. Dan, Meshman, Vechev, and Yahav [82] propose a reduction of a program and a relaxed memory model to a program under sequential consistency. The reduced program over-approximates the behavior of the given program running on the provided relaxed memory model. Then an off-the-shelf state of the art analyzer is used to analyze the reduced program. Chaki, Gurfinkel, Kong, and Strichman [58] sequentialize a time-bounded periodic program<sup>2</sup> and then use CBMC [69] for model checking.

*Regression.* The task of checking regressions can also be reduced to a verification task that can be handled by an off-the-shelf verifier. Differential assertion checking [155] checks for regressions concerning a given set of assertions. The approach considers the previous version of a program as a specification and checks the newer version against it. It reduces the problem to a safety-checking task by producing a composed program from the newer and the previous version of the program. PEQTEST [136] tests for functional equivalence after refactoring. The technique produces a new program that encodes the equivalence and then uses an off-the-shelf test generator to find differences.

*Loop transformations.* IMPULSE [149] first reduces a C program and then verifies it using CBMC [69] or LAWI [179] as an off-the-shelf verifier. The proposed reduction under-approximates the loop behavior, thereby, preventing the subsequent verifier from exploring numerous spurious counterexamples. Kroenig, Lewis, and Weissenbacher [150] provide a reduction that enables the use of bounded model checking for sound verification. This reduction produces a shallow program that preserves the reachable states of the original program allowing the use of bounded model checking for safety verification. The technique employs loop acceleration [44] and then uses trace abstraction [124] to remove redundant execution paths.

Loop shrinking [152] is a program-to-program transformation for reducing bounds of loops that process arrays. The output program is an over-approximation of the original one, resulting in reducing the complexity of the verification problem. After this transformation an off-the-shelf model checker is used for verification.

<sup>2</sup> A set of asynchronous tasks that are executed periodically. Each task is scheduled according to its priority.

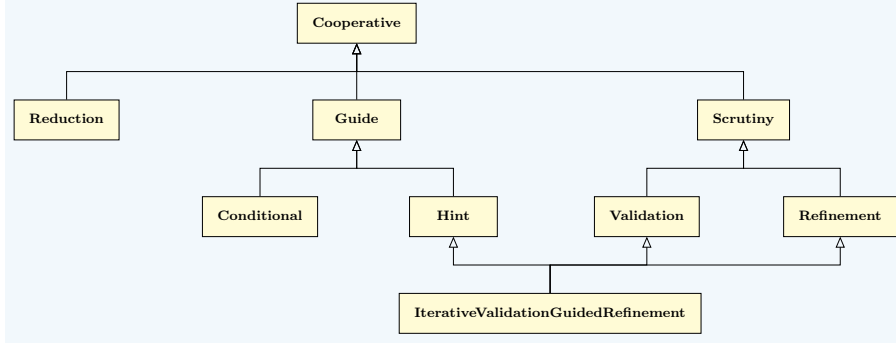


Fig. 3: Classes of cooperative verification techniques

*Program Simplification.* TRIMMER [95] is a program simplification technique that generates an *equi-safe* program but with fewer execution paths. An off-the-shelf safety checker can then be used to verify this program.

QUERYMAX [230] is a tool that aims to increase the precision of a whole-program analysis without affecting its speed. It computes the part of the library that is relevant for the analysis of the given program and specification and provides it as an input to an off-the-shelf verifier. As a result, the verifier is not required to over-approximate the library and hence can produce results faster without losing precision.

Simic, Inverso, and Tribastone [217] propose a transformation to reduce the problem of checking if a given error bound can be violated in a code with fixed-point arithmetic to a reachability problem. The transformation is implemented in the CSEQ [96] and the verifier used for reachability check is CBMC [69].

#### 4.2 Guide: Artifact Acts as a Guide

In this class of cooperative techniques, an analyzer produces an artifact that guides the next analyzer in verification. This artifact could be a summary of the verification work that has been completed, a hint to guide state space exploration, or a counterexample to guide the refinement of an abstract.

##### 4.2.1 Conditional: Artifact Represents a Condition

Conditional verification allows complementary analyses to cooperate. A conditional analyzer analyzes the program under an assumption which is termed as a *condition*. A condition could be provided as input or can be inferred by a formal analysis [24]. The key idea is to make progress in solving the verification task even if

by making some compromises and explicitly document them by the means of a condition. This allows the later analyzers to concentrate only on these compromises instead of solving the complete verification task.

Conditional model checkers [24] output a condition, usually a state predicate, if they cannot finish the verification within the allocated resources. This condition summarizes the work done by the model checker. Conditional model checkers can also take a condition, in addition to the given program and specification, as an input and process it to save effort by focusing the search only on the unexplored state space.

Czech, Jakobs, and Wehrheim [80] propose a cooperation between a conditional model checker and a tester. Their approach uses the exported condition to generate a residual program which is then given as input to a test generator.

Christakis, Müller, and Wüstholtz [65] combine static analysis and testing. The idea is to let a verifier progress, even if by making some unsound assumptions, and later test these unsound assumptions. The authors later improved this technique by adapting dynamic symbolic execution to abort tests leading to verified executions [64].

COVERITEST [27] iteratively applies conditional model checking and generates tests from the counterexample traces produced by model checking. It encodes test goals as targets for reachability specifications and executes various analyses with specific time budgets. At each step, one analysis executes and generates counterexample traces if it finds a violation of the reachability specification. COVERITEST then generates tests from these traces. When the time budget of the executing analysis is exhausted, a condition representing the explored state space is passed on to the next conditional model checker. This sequence continues until tests are generated for all the test goals.

Conditional testing [33] ports the idea of conditional model checking to testing. A conditional tester, in addition to generating a test suite, outputs a set of test goals that are covered by the test suite. This information can be used by another conditional tester to save effort.

Abstraction driven concolic testing [81] iteratively combines concolic testing and model checking. Concolic testing generates tests covering as many goals as it can in the given time budget. Then, the model checker is executed with the remaining goals marked as error locations. The model checker tries to reach as many error locations as it can and generates tests from the error traces. It removes goals from the set that it finds unreachable. It then refines the abstraction with respect to the remaining goals. The program is reduced using this refined abstraction and given again to the concolic tester. The combination uses CREST [51] for concolic testing and CPACHECKER [31] for model checking. DES-CEGAR [224] use a combination of dynamic symbolic execution and CEGAR-based model checking for data flow testing [128, 200].

Lanzinger, Weigl, Ulbrich, and Dietl [158] aim to complement type checking with deductive verification. The technique assumes the use of advanced type systems. In the first step, the type checker executes and checks the program. In case type checking fails, it outputs partial typing information and errors, which are then encoded in the given program as specification clauses, and the instrumented program is given to a deductive verifier.

MPI-SV [244] combines symbolic execution and model checking for the verification of message passing interface [117] programs. The technique iteratively calls a symbolic executor and a model checker for communicating sequential processes. The symbolic executor explores the path and generates a test if a violation is found. Otherwise, MPI-SV converts the violation-free path to a communicating sequential processes model and calls the model checker. Again, a test is generated if a violation is found, otherwise, the program is pruned and fed back to the symbolic executor.

#### 4.2.2 Hint: Artifact Acts as a Hint for the Verification Process

In this set of cooperative techniques, an analyzer produces a hint that guides the subsequent verification process. This hint could serve as either a starting point for the verification process or provide computed knowledge that guides a verification actor in exploration. In contrast to the class *conditional*, the first analyzer in this case is playing a supporting role. In conditional, the first analyzer tries to solve the verification task by

itself, and the second analyzer is executed only if it is not able to do so.

*Testing.* MAP2CHECK [206] uses claims from a bounded model-checker to generate assertion for test cases. The authors provide an implementation based on ESBMC [73] and CUNIT testing framework [1].

*Concurrent Software.* AUTOCONTEST [226] is a coverage-driven approach to generate test code for concurrent classes. It iteratively executes a sequence of three steps. The first step computes coverage requirements and generates sequential tests. The second step assembles concurrent tests from the sequential ones, and the third step explores newly covered interleavings and selects the failing tests.

Assertion guided abstraction [153] presents a cooperation between static and dynamic analysis techniques. The employed static analysis calculates predicate dependence and the dynamic analysis (stateless model checking [110]) uses this knowledge to reduce the interleaving space to be explored.

Deng, Zhang, and Lu [84] propose a new coverage metric to approximate the overlapping of interleavings for a set of inputs. The first step in their combination uses this coverage metric to select test inputs to give to off-the-shelf data-race or atomicity-violation detectors.

NARADA [210] is an approach that generates tests triggering race condition based on the given seed of sequential tests. It is a three-stage approach. The first stage analyzes the execution traces of a sequential seed test-suite, the second stage uses this information to generate constraints applicable to method calls, and the third stage synthesizes racy tests.

Hong, Ahn, Park, Kim, and Harrold [131] propose a technique that tries to achieve high coverage for concurrency testing for a given test case and program. It is a combination of two phases: estimation and testing. The estimation phase produces synchronization-pair coverage requirements. The test phase executes the test, measures coverage, and iteratively changes the thread schedule to achieve more coverage.

*Invariant Injection.* Beyer, Dangl, and Wendler [18] present a technique to boost  $k$ -induction [88] with the help of a data-flow-analysis-based invariant generator. The generated invariants are continuously injected in the  $k$ -induction engine, which uses them to strengthen the hypothesis for the next iteration. COVEGI [123] employs various off-the-shelf tools to generate invariants which are used by the main verifier to solve the verification problem.

PEGASUS [218] generates invariants for deductive verification of hybrid and continuous systems. It is integrated with the theorem prover KEYMAERA X [101].

KEYMAERA X asks PEGASUS for invariants when it cannot solve a continuous safety verification problem. KEYMAERA X validates the returned invariants and proof hints and uses them to find the safety proof.

ICE/FIRE [43] is based on the cooperation of inductive checking engine (ICE) and finite reachability checking engine (FIRE). Both these components pass information to each other and are executed iteratively. ICE tries to decide the safety problem using *k-induction* [88]. If it is not able to find proof, it asks FIRE a bounded reachability query. FIRE either returns a bounded invariant which is used to guide ICE to prove safety, or a trace. This is repeated until the problem is solved.

Bouillaguet, Bobot, Sighireanu, and Yakobowski [46] present a cooperation between a static analysis supporting pointer analysis and a deductive verification engine based on the first-order logic. The static analyzer produces state invariants and sound memory partitioning. The deductive verifier uses this information to produce verification conditions, which are discharged by automatic solvers. This approach is implemented inside the FRAMA-C [145] platform.

*Miscellaneous.* SEGATE [220] implements a test generation technique for structured string input combining static analysis and test generation. Static analysis analyzes the implementation and infers a regular expression to be given as input to the test generator. This inferred regular expression is given along with the regular expression (human-annotated or machine-generated) based on the specification of the test generator. The test generator generates tests based on the difference between these two regular expressions.

Lee, Lee, and Ryu [160] present a static analysis technique for software where different parts of the software are written in different languages. The analysis is based on the cooperation between static analyzers of the host language and the guest language. The technique first extracts the summaries of the code written in the guest language, then converts them to the summaries in the host language, injects them into the host language code, and finally uses a static analyzer of the host language to analyze the modified program.

IDISCOVERY [248] combines invariant generation and symbolic execution to improve the quality of the generated invariants. It is a sequence of four actors: invariant generator, assertion synthesizer, instrumentor, and symbolic executor. It first generates candidate invariants based on the given test suite using DAIKON [92], then it generates assertions based on the candidate invariants, then it instruments the assertions into the program, and then uses symbolic execution to generate test inputs which are then fed back into DAIKON again.

Zhang, Groce, and Alipour [247] present the idea of minimizing and prioritizing a set of given seed test cases to improve symbolic execution. The test cases are prioritized to maximize the efficiency of exploration.

Busse, Gharat, Cadar, and Donaldson [52] present an approach combining static analysis with dynamic symbolic execution. The static analysis produces a partial trace annotated with conditions it deems important. Dynamic symbolic execution takes this partial trace as a guide for its search. This may allow the bug to be confirmed quickly.

PERIOD [237] implements a controlled concurrency testing technique. First, a schedule generator generates schedules to be tested for potential bugs, then these schedules are executed to check for the presence of bugs, and then the output of the execution is analyzed to guide the generation of schedules in the next iteration.

Lohar, Jeangoudoux, Sobel, Darulova, and Christakis [170] present a two-phase cooperative approach for floating-point analysis. The first phase is based on a dynamic analysis that identifies small parts of the program performing complex numerical computations called numerical kernels and the corresponding input ranges. The second phase runs off-the-shelf verifiers to verify these kernels. The first phase uses abstract interpretation to infer input ranges. If abstract interpretation cannot infer finite ranges then it uses fuzzing, which might produce unsound results.

#### 4.3 Scrutiny: Artifact is Scrutinized

In this class of cooperative techniques, an analyzer produces an artifact and another analyzer is employed to scrutinize or examine it. As an outcome, the artifact can be validated or rejected, filtered, or refined.

##### 4.3.1 Validation: Artifact is Validated by a Subsequent Actor

In this class of cooperative techniques, an analyzer produces, in addition to the result of analysis, an artifact as potential evidence for the result. Another analyzer is then employed to examine and scrutinize the potential evidence. The scrutinizing analyzer either validates the potential evidence, or if the artifact is a set of potential evidences then the actor filters out the spurious ones.

*Alarm Validation.* Static analysis tools can produce alarms quickly. However, they are also known to produce false alarms because they analyze an abstract model of the given program. To tackle this problem, another analyzer can be used to validate these alarms.

One such work uses a GraphML [49] based format to export the alarms and then employ another analyzer to validate them [17]. Li, Chen, Wang, and Xu [167] propose a cooperative technique to validate memory leak warnings. First, a static analysis tool generates memory leak warnings. In the next step, these warnings are instrumented in the given program. Then the technique uses a modified concolic testing engine to generate tests for the instrumented program. The runtime trace produced by executing these tests is used to validate and classify the warnings.

*Concurrent Software.* Analysis tools can detect potential races in software. These races can only be triggered when some conditions are met, e.g., methods are called in specific order, or interrupts occur at specific points. Specialized analysis tools can try to trigger these potential races and validate their feasibility.

SDRACER [236] uses a combination of static analysis and symbolic execution to generate test input for potential races and dynamically validates these potential races. VINDICATOR [207] uses an imprecise analysis to predict possible races and then a validation step to filter out the false positives, thus creating a sound analysis. Ganai [102] presents a predictive testing approach that generates witness schedules for potential data races and then employs an independent component to validate if they can indeed cause a data race. SIMRACER [245] is a tool for detecting process-level races. Similar to the above techniques, it first produces potential races based on the execution of existing tests and then validates them.

EXCEPTIONNULL [93] analyzes test executions to predict interleavings that can cause null-pointer dereferences and then schedule them to validate the bug. INTATOM [163] detects interrupt atomicity violation bugs. It identifies candidates for atomicity violation using data flow analysis and then checks their feasibility.

*Android.* Event-based race detectors for Android apps can produce false positives. ERVA [132] applies a validation phase for the potential event-based race reports. It further classifies the confirmed races into benign or harmful.

*Regression.* Verification-aided regression testing [195] is a cooperative approach that aims to augment regression testing. In the first phase, it uses Daikon [91, 92] to infer properties from a test suite for the base version of the program and verifies them using a model checker. In the second phase, it uses these properties to identify regressions. The technique also makes use of the tests created for the changes made to the program to filter out the properties intentionally violated by the change.

RECONTEST [227] identifies the interleavings that are caused by the changes in the code, selects those that have the potential to cause regression, and finally, validate them to report any regression bugs found.

*Infer and Validate.* Zhu, Nori, and Jagannathan [249] propose a combination of random testing and template inference to infer array invariants. They then use a verifier to validate these invariants.

Neural termination analysis [108] uses neural networks as ranking functions for termination analysis. In the first step, it generates test inputs and collects execution traces for these test inputs. The second step uses the execution traces to train neural networks that behave like a ranking function over the collected execution traces. The authors use the term neural ranking functions for such neural networks. In the last step, a verifier tries to prove termination with the help of the neural ranking function. The verifier encodes both the program and the neural ranking function symbolically and uses an SMT solver to decide termination.

*Proof Validation.* A verifier can also produce a safety proof in case it finds the program to be safe. Such proofs can be exported in a GraphML [49] based format and then validated by another analyzer [16]. In the first step, a verifier exports a correctness witness as proof when it proves the program to be safe. In the second step, another analyzer validates the proof.

#### 4.3.2 Refinement: Artifact is Refined by the Subsequent Actor

Static analyzers are imprecise (even unsound) but fast. This class of cooperative techniques aims to benefit from the speed of the fast but potentially imprecise analysis tools by refining the artifacts produced by them.

*Alarm Refinement.* Residual investigation [165] refines the errors reported by static analysis. Instead of simply replaying the error, it tries to identify general conditions under which the error reported by static analysis is confirmed. As a result, it acts as a filter for the errors reported by the static analysis.

PEAHEN [54] is a combination-based technique for detecting deadlocks. The first step constructs a context-insensitive lock-graph of the program. The second step employs three refinement techniques for refining deadlock cycles in the lock graph.

*Concurrent Software.* DOUBLECHECKER [40] combines an imprecise but fast and a precise but slow analysis to check for atomicity violations. The imprecise analysis over-approximates the dependence edges, hence producing a set of potential cycles. The precise analysis refines these potential cycles.



*Android*. TCM [146] first uses a test generator (ANDROFRAME [147]) to generate test cases, then minimizes this test suite, and then applies mutation on the minimized test suite to get a richer test suite.

#### 4.3.3 Iterative Validation Guided Refinement

Computing a result with the available insufficient information, validating it, and then using the results of the validation to guide the refinement of the result is a sequence that is well known in software verification [67]. This sequence is generally executed iteratively until the result is found or the computation runs out of resources.

*Invariant Synthesis*. Neider, Garg, Madhusudan, Saha, and Park [185] present a data-driven invariant synthesis approach that is guided by the non-provability information provided by the verifier. It is an iterative process where the invariant generator proposes invariants, and the verifier tries to use them to verify the program. In case the verifier is unsuccessful, it communicates the non-provability information to the invariant generator, which is used to refine the invariant in the next iteration.

Garg, Neider, Madhusudan, and Roth [104] present a cooperative technique for generating inductive invariants by combining a verifier and an invariant learner. A decision-tree-based learning algorithm synthesizes and proposes invariant hypotheses, and a verifier checks them. In case the invariant is invalid, the verifier returns a counterexample, which is then used to revise the hypothesis by the learner.

Danger invariants [83] are a compact representation of counterexample traces. The proposed technique is an iterative process of synthesizing a danger invariant, verifying it using an off-the-shelf verifier, and then feeding back the counterexample from the verifier to the invariant synthesizer that uses it to refine the invariant.

NUMINV [186] infers candidate invariants using algorithms from DIG [188] and uses KLEE to validate them.

ZILU [164] uses learning and verification to generate loop-invariants. It collects states by executing randomly generated test cases, and generates a candidate invariant from these states. It uses selective sampling to refine the candidate and then uses a verifier to check it. In case of failure, the counterexample is used to refine the candidate invariant.

SYMINFER [187] generates symbolic states using a symbolic executor, generates concrete states from them and uses DIG [188] to generate candidate invariants, and checks if the candidate invariants are consistent with respect to the symbolic states. This process is repeated till the invariants can be refuted or timeout occurs. This approach might produce unsound invariants.

IMPLCHECK [205] uses various off-the-shelf tools to iteratively strengthen the set of inductive invariants until the program is found to be safe. Initially, it employs off-the-shelf invariant generators based on syntax-guided synthesis [8] or data learning to generate the initial set of candidate invariants. Then, it uses a HOUDINI [98] based algorithm to compute an inductive subset of invariants from the initially computed set of candidate invariants. It then attempts to prove the program safe with the help of this inductive subset of invariants. If it fails to prove the program safe, it selects a non-inductive invariant from the initial set of candidate invariants and weakens it. This process is repeated until either the program is proven safe or the algorithm runs out of candidate invariants to check.

CODE2RELINV [235] is a relational invariant generator based on a combination of an invariant synthesizer and a program verifier. The synthesizer is based on reinforcement learning [140] and uses syntax-guided synthesis [8]. The program verifier validates the synthesized candidate invariants and provides the synthesizer with feedback in case a given candidate invariant is found to be non-inductive or insufficient. The synthesizer uses logical reasoning to prune search space and reinforcement learning to prioritize search.

*Abstraction Refinement*. UFO [5] brings together over-approximation driven and under-approximation driven analysis approaches to model checking. It executes over-approximation-driven analysis to construct a completely explored abstract reachability graph. If this graph contains an error location, then the under-approximation-driven analysis is used to check the feasibility of the potential error and refine it. This combination is executed iteratively.

C-CEGAR [22] decomposes CEGAR into three off-the-shelf components. The three components are: abstract model explorer that generates potential counterexamples, feasibility checker that checks these counterexamples, and precision refiner that uses the infeasible counterexample to refine the precision of the abstract model. This process continues iteratively until the program is proven correct or a bug is found.

Abdulla, Atig, and Diep [3] propose a technique for program verification that first constructs an abstract model of the given program, then uses an off-the-shelf model checker for model checking, and then refines the abstraction using the counterexample returned by the model checker. This sequence is executed iteratively until either the model checker finds the abstract model to be safe or a feasible counterexample is found.

*Inferring Loop Bounds*. TPT [192] and DDLTERM [242] use machine-learning-based techniques to infer loop

bounds, validate them, and use the counterexample information to refine the candidate loop bound in the next iteration. TPT [192] uses the existing test-suite for a sequential program to infer loop bounds and employs linear regression for inference. DDLTERM [242] uses various data-driven algorithms to infer different kinds of loop bounds. It can infer simple, conjunctive, and lexicographic bounds.

CoMuS [201] reduces the verification of a concurrent program to a series of verification problems for a sequential verifier. It considers one thread for verification and abstracts all the remaining threads as *environment*. It then uses an off-the-shelf model checker for verification and checks if the counterexamples produced by the model checker are feasible. If not, then it refines the *environment*, modifying the sequential program as a result. This process is repeated until the verification succeeds or a valid counterexample is found.

## 5 Cooperative Verification Actors and Artifacts

### 5.1 Cooperative Actors

During our review, we found some articles that did not present a cooperative verification technique themselves but rather presented algorithms and tools that can act as a cooperative actor. These techniques could be used in combination with other cooperative actors to construct a cooperative combination.

#### 5.1.1 Reducers

TINA [202] reduces the given C program containing inline assembly code to a semantically equivalent pure C program. Most off-the-shelf verifiers for the C language do not support inline assembly code. TINA enables a user to use an off-the-shelf verifier for a program. One can create a cooperative combination by chaining TINA and an off-the-shelf verifier.

DROIDRA [166] reduces the code of an Android app such that it does not contain reflective calls. The results could be analyzed by static analyzers which do not support reflection.

A conditional model checking [24] problem can be reduced that of model checking with the help of reducers [28]. The proposed reducer takes a program and a condition produced by a conditional model checker representing the explored state space and produces a program that contains only the unexplored parts of the state space. Then, an off-the-shelf model checker can be used for verification.

#### 5.1.2 Validators

A validator is an analyzer that validates the result produced by another analyzer. For example, a verification tool can produce a witness along with the verdict when it succeeds in computing the result. A validator in this case tries to reconstruct the solution of the verification problem with the help of the witness, and as a result, confirms or rejects the given witness.

METAVAL [36] transforms a given validation problem into a verification problem. It embeds the original program with hints for validation available in the given witness. Then, it employs an off-the-shelf verifier for verification. The verification result of the transformed program corresponds to the validation of the initial program and the provided witness.

NITWIT [250] is an interpretation-based validator for alarms. It combines a C interpreter and a witness automaton constructed from the violation witness and tries to reach the error label.

Lee, Lee, and Yi [161] present a sound clustering algorithm for alarms produced by abstract-interpretation-based static analysis techniques. A transformer-based [199, 225] machine learning approach has also been used to identify false alarms [143].

#### 5.1.3 Refiners

A refiner refines the artifact produced by an analyzer.

*Refining alarms.* Another approach refines the output of a static analyzer using call graph pruning [229]. It uses a machine-learning-based classifier to select the edges to prune in the generated call graph. This approach reduces the false positives but also reduces the true positives. AUTOPRUNER [159] prunes call graphs to eliminate false positives using both statistical semantic and structural analysis.

AVFILTER [246] filters the error predictions made by an over-approximating analysis. It improves the precision of an over-approximate analysis.

*Test suite reduction.* Test suite reduction can be seen as a refinement. One can create a cooperative combination by using a test suite generator that might produce some redundant test cases and then calling a tool that reduces the generated test suite to produce a smaller test suite. During our review, we found articles presenting test suite reduction techniques that can act as cooperative actors. FLOWER [115] uses network maximum flow,

*Reducing assertions.* One approach to software verification is to automatically generate assertions for a program and then checking them using software analysis tools.



However, assertion generators can generate many redundant assertions. Fediyukovich, D’Iddio, Hyvärinen, and Sharygina [94] propose an algorithm that finds out the redundant assertions, thereby reducing the number of assertions to be checked by a model checker.

#### 5.1.4 Invariant Generators

A tool that generates invariants can also act as a cooperative actor. It can feed invariants to a verification algorithm, such as  $k$ -induction [88]. PIE [194] is a learning-based tool for inferring preconditions that can be used to generate candidate invariants. HOLA [87] implements a method for generating inductive invariants based on a combination of verification condition generator and abductive inference to guess candidate invariants.

#### 5.1.5 Abstraction Generators

Naik, Yang, Castelnovo, and Sagiv [184] propose a technique that uses concrete tests to generate abstractions that can then be used by a static analysis tool.

### 5.2 Cooperative Artifacts

Witnesses [16, 17] (as used in SV-COMP) are based on GraphML [49]. Witnesses can represent either error traces or proof of safety. Witnesses in this format have found acceptance in the verification community and have been used in SV-COMP for a few years.

A succinct representation of concurrent trace sets [121] aims to reduce the size of an artifact representing a set of concurrent traces. This succinct representation can be used to exchange information.

PART<sub>PW</sub> [135] is a technique that can generate proofs from the results of complementary analyses. It is used in conjunction with conditional model checking [24] to generate proofs in addition to alarms.

## 6 Threats to Validity

We conducted a literature review that provides an overview and classification of cooperative verification techniques. We now discuss the threats we identified in our work.

*Number of authors.* This study was conducted by two authors. This type of study would have benefited with the involvement of more authors. However, the authors of this study are experts in the field of cooperative verification: one author is a pioneer in this area and the other has been working on it for a few years. We claim that this fact mitigates the threat a little. Additionally, we

restrained ourselves from making a stronger claim: it is an exploratory survey, not an exhaustive one. We expect the results of this survey to be useful in developing an understanding of cooperative verification.

*Selection of publications.* The corpus of literature we reviewed is a small subset of all the available scientific literature as we limited ourselves to the selected publications and time duration. As a result, some articles discussing cooperative verification techniques might have been excluded. However, the publications and the period we selected gave us a corpus of publications sufficient to mine a representative set of cooperative verification techniques for developing an overview and a classification.

*Filtering process.* We executed our search query only on the *titles* and *abstracts*. Since the search was conducted only on a small portion of the article, it has the potential to leave out many articles. We chose this strategy because our search terms are too permissive. We could not leave out terms like *test* and *verify* because these are important for software verification. This choice made our filter weak when we tried to search for the complete text of chosen publications. Moreover, we found it reasonable to assume that a publication about software verification would contain at least one of the terms allowed by our search query in the title or the abstract.

*Bugs and oversight.* A part of the filtering process was automated. There is a potential for bugs to be present in our scripts. During the development of these scripts, we manually validated the results on a small sample size. We executed the scripts on the complete corpus when we were confident about the correctness of our scripts.

Another part of our filtering process was manual. There is a potential for oversight. We provide an artifact containing the data corresponding to the different stages of our methodology. The data is available for review.

*Definition.* We developed a definition for cooperative verification that provides a decision process for identifying a cooperative technique. This definition formalizes the intuitive notion of a cooperative technique from various articles [24, 29, 33, 38, 65, 123]. One can argue about the level of permissiveness of our definition. Some might find it too restrictive, some too permissive. However, this is a threat that accompanies developing definitions.

*Identification of an approach as cooperative.* Although we had a decision procedure based on our definition, we still faced challenges in identifying some cooperative approaches. Publications from different areas give importance to different aspects of an approach: some publications focus more on theory, while others more

on implementation. It was easier to decide on an article that focused more on the implementation aspect than the one focusing more on theory. We chose to be permissive when in doubt instead of being restrictive. As a result, the set of cooperative techniques we have identified might be an over-approximation.

## 7 Discussion

### 7.1 Insights

Our classification of cooperative techniques resonates with the ideas commonly used during software analysis: *reduction* of a problem to another problem, *validation* of the computed result, *refinement* of an abstraction to reach the level of precision required to solve the verification task, *conditionality* to prune away the uninteresting part of the state space, and *guide* the state space exploration using hints.

Our survey shows that researchers have been working on the specific parts of the above-mentioned commonly performed tasks during verification, and combining them. With this survey, we aim to push forward the idea of decoupling the tools and development of formats for information exchange.

### 7.2 Use of clearly specified interfaces in the reviewed literature

The cooperative approaches benefit from information exchange in a standardized format. This allows for independent development of ideas by relieving dependency on a specific technology or a tool. It allows to reuse the available tools and technologies, thereby, saving time and effort,

Ideally, one should be able to discern the following information from such a paper: (1) Actors or components, (2) Kind and mechanism of information exchange, and (3) Arrangement of actors (e.g., are they executed one after another or in a loop).

However, this information was not highlighted in many papers. This survey would have contained only a handful of papers if we would have been strict in our interpretation. We think that highlighting the above mentioned information will advance development of cooperative techniques and foster reuse.

### 7.3 Benefits of using standardized interfaces

Development and use of standardized interfaces for information exchange for the purpose of verification will

have the following benefits: (1) Easy identification of actors (reusable and replaceable components) and artifacts (means of information exchange), (2) Faster development of new cooperative ideas because of ease of reuse of actors and artifacts, (3) Ease of reuse will foster easily portable techniques, and (4) Replaceable components will allow to measure effect of a specific type of actor in a combination.

### 7.4 Call to action

Information exchange is at the heart of cooperative verification. We need a consensus on which information to share and how. The development of standardized formats for information exchange would help propel this field. This development would also make it easier to define the interface of a cooperative verification actor. We envision this as community effort which would benefit from experience of experts with different backgrounds.

## 8 Conclusion

Cooperative verification brings together different approaches to combinations that can achieve more than just the sum of the individual approaches. It requires us to consider verification approaches as components with clear interfaces. Those interfaces are not (yet) standardized and the roles of the participating objects were not yet classified. We have used the notions of actors and artifacts and tried to characterize what the various roles are. We have presented an overview of the verification techniques that we identified as cooperative and a classification of these techniques. We hope that our review helps to develop a deeper understanding of cooperative verification. As the field of cooperative verification advances, we expect to see more and more combination approaches for software verification being presented as combinations of cooperating actors that exchange information.

**Data-Availability Statement.** All data that we processed for this work are available at Zenodo [30].

**Funding Statement.** This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY).

**Acknowledgements.** We thank Stefan Winter for valuable discussions about the methodology and sharing the data for ACM publications from a previous project [239]. We thank Thomas Bunk for proof-reading the manuscript and providing valuable feedback.

## References

1. C unit testing framework. <https://cunit.sourceforge.net/>. Accessed: 2025-05-18
2. Computing Research & Education Conference Portal. <http://portal.core.edu.au/conf-ranks/>. Accessed: 2023-04-01
3. Abdulla, P.A., Atig, M.F., Diep, B.P.: Counter-Example Guided Program Verification. In: Proc. FM, pp. 25–42. Springer (2016). [https://doi.org/10.1007/978-3-319-48989-6\\_2](https://doi.org/10.1007/978-3-319-48989-6_2)
4. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995). ISBN: 0-201-53771-0
5. Albarghouthi, A., Gurfinkel, A., Chechik, M.: From under-approximations to over-approximations and back. In: Proc. TACAS, LNCS 7214, pp. 157–172. Springer (2012). [https://doi.org/10.1007/978-3-642-28756-5\\_12](https://doi.org/10.1007/978-3-642-28756-5_12)
6. Alglave, J., Kröning, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: Proc. ESOP, LNCS 7792, pp. 512–532. Springer (2013). [https://doi.org/10.1007/978-3-642-37036-6\\_28](https://doi.org/10.1007/978-3-642-37036-6_28)
7. Alt, L., Asadi, S., Chockler, H., Even-Mendoza, K., Fedayukovich, G., Hyvärinen, A.E.J., Sharygina, N.: HiFrog: SMT-based function summarization for software verification. In: Proc. TACAS, LNCS 10206, pp. 207–213 (2017). [https://doi.org/10.1007/978-3-662-54580-5\\_12](https://doi.org/10.1007/978-3-662-54580-5_12)
8. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Proc. FMCAD, pp. 1–8. IEEE (2013). <https://doi.org/10.1109/FMCAD.2013.6679385>
9. Alur, R., Henzinger, T.A.: Reactive modules. In: Proceedings of the 11th Annual Symposium on Logic in Computer Science, pp. 207–218. IEEE (1996)
10. Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional verification by learning assumptions. In: Proc. CAV, pp. 548–562. Springer (2005). [https://doi.org/10.1007/11513988\\_52](https://doi.org/10.1007/11513988_52)
11. Baier, D., Beyer, D., Friedberger, K.: JAVASMT 3: Interacting with SMT solvers in Java. In: Proc. CAV. Springer (2021). [https://doi.org/10.1007/978-3-030-81688-9\\_9](https://doi.org/10.1007/978-3-030-81688-9_9)
12. Baldoni, R., Coppa, E., D’Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3), 50:1–50:39 (2018). <https://doi.org/10.1145/3182657>
13. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al.: cvc5: A versatile and industrial-strength smt solver. In: Proc. TACAS, pp. 415–442. Springer (2022). [https://doi.org/10.1007/978-3-030-99524-9\\_24](https://doi.org/10.1007/978-3-030-99524-9_24)
14. Beckman, N., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: Proc. ISSTA, pp. 3–14. ACM (2008). <https://doi.org/10.1145/1390630.1390634>
15. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq’Art: the calculus of inductive constructions. Springer (2004). <https://doi.org/10.1007/978-3-662-07964-5>
16. 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>
17. 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>
18. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV, LNCS 9206, pp. 622–640. Springer (2015). [https://doi.org/10.1007/978-3-319-21690-4\\_42](https://doi.org/10.1007/978-3-319-21690-4_42)
19. Beyer, D., Friedberger, K.: Domain-independent multi-threaded software model checking. In: Proc. ASE, pp. 634–644. ACM (2018). <https://doi.org/10.1145/3238147.3238195>
20. Beyer, D., Friedberger, K.: Domain-independent interprocedural program analysis using block-abstraction memoization. In: Proc. ESEC/FSE, pp. 50–62. ACM (2020). <https://doi.org/10.1145/3368089.3409718>
21. 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](https://doi.org/10.1007/978-3-319-10575-8_16)
22. Beyer, D., Haltermann, J., Lemberger, T., Wehrheim, H.: Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR. In: Proc. ICSE, pp. 536–548. ACM (2022). <https://doi.org/10.1145/3510003.3510064>
23. 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). <https://doi.org/10.1007/s10009-007-0044-z>
24. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE. ACM (2012). <https://doi.org/10.1145/2393596.2393664>
25. Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy shape analysis. In: Proc. CAV, LNCS 4144, pp. 532–546. Springer (2006). [https://doi.org/10.1007/11817963\\_48](https://doi.org/10.1007/11817963_48)
26. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV, LNCS 4590, pp. 504–518. Springer (2007). [https://doi.org/10.1007/978-3-540-73368-3\\_51](https://doi.org/10.1007/978-3-540-73368-3_51)
27. Beyer, D., Jakobs, M.C.: CoVeriTest: Cooperative verifier-based testing. In: Proc. FASE, LNCS 11424, pp. 389–408. Springer (2019). [https://doi.org/10.1007/978-3-030-16722-6\\_23](https://doi.org/10.1007/978-3-030-16722-6_23)
28. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE, pp. 1182–1193. ACM (2018). <https://doi.org/10.1145/3180155.3180259>
29. Beyer, D., Kanav, S.: CoVeriTEAM: On-demand composition of cooperative verification systems. In: Proc. TACAS, LNCS 13243, pp. 561–579. Springer (2022). [https://doi.org/10.1007/978-3-030-99524-9\\_31](https://doi.org/10.1007/978-3-030-99524-9_31)
30. Beyer, D., Kanav, S.: Reproduction package for STTT submission ‘Cooperative verification: A literature review’. Zenodo (2023). <https://doi.org/10.5281/zenodo.7838608>
31. Beyer, D., Keremoglu, M.E.: CPACHECKER: A tool for configurable software verification. In: Proc. CAV, LNCS 6806, pp. 184–190. Springer (2011). [https://doi.org/10.1007/978-3-642-22110-1\\_16](https://doi.org/10.1007/978-3-642-22110-1_16)
32. Beyer, D., Lee, N.: The transformation game: Joining forces for verification. In: Principles of Verification: Cycling the Probabilistic Landscape - Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part III, LNCS, vol. 15262, pp. 175–205. Springer (2024)
33. Beyer, D., Lemberger, T.: Conditional testing: Off-the-shelf combination of test-case generators. In: Proc. ATVA, LNCS 11781, pp. 189–208. Springer (2019). [https://doi.org/10.1007/978-3-030-31784-3\\_11](https://doi.org/10.1007/978-3-030-31784-3_11)
34. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). <https://doi.org/10.1007/s10009-017-0469-y>

35. Beyer, D., Podolski, A.: Software model checking: 20 years and beyond. In: Principles of Systems Design, LNCS 13660, pp. 554–582. Springer (2022). [https://doi.org/10.1007/978-3-031-22337-2\\_27](https://doi.org/10.1007/978-3-031-22337-2_27)
36. Beyer, D., Spiessl, M.: METAVAL: Witness validation via verification. In: Proc. CAV, LNCS 12225, pp. 165–177. Springer (2020). [https://doi.org/10.1007/978-3-030-53291-8\\_10](https://doi.org/10.1007/978-3-030-53291-8_10)
37. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. arXiv/CoRR 1905(08505) (2019). <https://arxiv.org/abs/1905.08505>
38. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1), LNCS 12476, pp. 143–167. Springer (2020). [https://doi.org/10.1007/978-3-030-61362-4\\_8](https://doi.org/10.1007/978-3-030-61362-4_8)
39. Bianchi, A., Corbetta, J., Invernizzi, L., Fratantonio, Y., Kruegel, C., Vigna, G.: What the app is that? deception and countermeasures in the android user interface. In: 2015 IEEE Symposium on Security and Privacy. IEEE (2015). <https://doi.org/10.1109/sp.2015.62>
40. Biswas, S., Huang, J., Sengupta, A., Bond, M.D.: DoubleChecker. In: Proc. PLDI. ACM (2014). <https://doi.org/10.1145/2594291.2594323>
41. Blackshear, S., Gorogiannis, N., O’Hearn, P.W., Sergey, I.: RacerD: compositional static race detection. PACMPL 2(OOPSLA), 1–28 (2018). <https://doi.org/10.1145/3276514>
42. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proc. PLDI, pp. 196–207. ACM (2003). <https://doi.org/10.1145/781131.781153>
43. Blicha, M., Hyvärinen, A.E.J., Marescotti, M., Sharygina, N.: A cooperative parallelization approach for property-directed k-induction. In: Proc. VMCAI, LNCS 11990, pp. 270–292. Springer (2020). [https://doi.org/10.1007/978-3-030-39322-9\\_13](https://doi.org/10.1007/978-3-030-39322-9_13)
44. Boigelot, B.: Symbolic methods for exploring infinite state spaces. Ph.D. thesis, Faculté des Sciences Appliquées de Université de Liège (1998)
45. Bouajjani, A., Boutglay, W.A., Habermehl, P.: Data-driven numerical invariant synthesis with automatic generation of attributes. In: Proc. CAV, pp. 282–303. Springer (2022). [https://doi.org/10.1007/978-3-031-13185-1\\_14](https://doi.org/10.1007/978-3-031-13185-1_14)
46. Bouillaguet, Q., Bobot, F., Sighireanu, M., Yakobowski, B.: Exploiting Pointer Analysis in Memory Models for Deductive Verification. In: Proc. VMCAI, pp. 160–182. Springer (2019). [https://doi.org/10.1007/978-3-030-11245-5\\_8](https://doi.org/10.1007/978-3-030-11245-5_8)
47. Braione, P., Denaro, G., Mattavelli, A., Pezzè, M.: Combining symbolic execution and search-based testing for programs with complex heap inputs. In: Proc. ISSA. ACM (2017). <https://doi.org/10.1145/3092703.3092715>
48. Braione, P., Denaro, G., Pezzè, M.: Jbse: a symbolic executor for java programs with complex heap inputs. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM (2016). <https://doi.org/10.1145/2950290.2983940>
49. Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., Marshall, M.S.: GraphML progress report. In: Graph Drawing, LNCS 2265, pp. 501–512. Springer (2001). <https://doi.org/10.1007/s100090050004>
50. Braun, V., Margaria, T., Weise, C.: Integrating tools in the ETI platform. STTT 1, 31–48 (1997). <https://doi.org/10.1007/s100090050004>
51. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: Proc. ASE, pp. 443–446. IEEE (2008). <https://doi.org/10.1109/ASE.2008.69>
52. Busse, F., Gharat, P., Cadar, C., Donaldson, A.F.: Combining static analysis error traces with dynamic symbolic execution (experience paper). In: Proc. ISSA. ACM (2022). <https://doi.org/10.1145/3533767.3534384>
53. 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)
54. Cai, Y., Ye, C., Shi, Q., Zhang, C.: Peahen: fast and precise static deadlock detection via context reduction. In: Proc. ESEC/FSE, pp. 784–796. ACM (2022). <https://doi.org/10.1145/3540250.3549110>
55. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O’Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM, LNCS 9058, pp. 3–11. Springer (2015). [https://doi.org/10.1007/978-3-319-17524-9\\_1](https://doi.org/10.1007/978-3-319-17524-9_1)
56. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about Datalog (and never dared to ask). IEEE Trans. Knowledge and Data Eng. 1(1), 146–166 (1989). <https://doi.org/10.1109/69.43410>
57. Chaki, S., Clarke, E., Sinha, N., Thati, P.: Automated assume-guarantee reasoning for simulation conformance. In: Proc. CAV, pp. 534–547. Springer (2005). [https://doi.org/10.1007/11513988\\_51](https://doi.org/10.1007/11513988_51)
58. Chaki, S., Gurfinkel, A., Kong, S., Strichman, O.: Compositional Sequentialization of Periodic Programs. In: Proc. VMCAI, pp. 536–554. Springer (2013). [https://doi.org/10.1007/978-3-642-35873-9\\_31](https://doi.org/10.1007/978-3-642-35873-9_31)
59. Chalupa, M., Richter, C.: BUBAAK-SPLit: Split what you cannot verify (competition contribution). In: Proc. TACAS (3), LNCS 14572, pp. 353–358. Springer (2024). [https://doi.org/10.1007/978-3-031-57256-2\\_20](https://doi.org/10.1007/978-3-031-57256-2_20)
60. Chen, Y.F., Clarke, E.M., Farzan, A., Tsai, M.H., Tsay, Y.K., Wang, B.Y.: Automated assume-guarantee reasoning through implicit learning. In: Proc CAV, pp. 511–526. Springer (2010). [https://doi.org/10.1007/978-3-642-14295-6\\_44](https://doi.org/10.1007/978-3-642-14295-6_44)
61. Cho, C.Y., D’Silva, V., Song, D.: BLITZ: Compositional bounded model checking for real-world programs. In: Proc. ASE. IEEE (2013). <https://doi.org/10.1109/ase.2013.6693074>
62. Choi, J., Jang, J., Han, C., Cha, S.K.: Grey-Box Concolic Testing on Binary Code. In: Proc. ICSE. IEEE (2019). <https://doi.org/10.1109/icse.2019.00082>
63. Choi, J., Kim, D., Kim, S., Grieco, G., Groce, A., Cha, S.K.: SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In: Proc. ASE. IEEE (2021). <https://doi.org/10.1109/ase51524.2021.9678888>
64. Christakis, M., Müller, P., Wüstholtz, V.: Guiding dynamic symbolic execution toward unverified program executions. In: Proc. ICSE, pp. 144–155. ACM (2016). <https://doi.org/10.1145/2884781.2884843>
65. Christakis, M., Müller, P., Wüstholtz, V.: Collaborative verification and testing with explicit assumptions. In: Proc. FM, LNCS 7436, pp. 132–146. Springer (2012). [https://doi.org/10.1007/978-3-642-32759-9\\_13](https://doi.org/10.1007/978-3-642-32759-9_13)
66. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MATHSAT5 SMT solver. In: Proc. TACAS, LNCS 7795, pp. 93–107. Springer (2013). [https://doi.org/10.1007/978-3-642-36742-7\\_7](https://doi.org/10.1007/978-3-642-36742-7_7)
67. 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). <https://doi.org/10.1145/876638.876643>
68. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT (1999). ISBN: 0262032708



69. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS, LNCS 2988, pp. 168–176. Springer (2004). [https://doi.org/10.1007/978-3-540-24730-2\\_15](https://doi.org/10.1007/978-3-540-24730-2_15)
70. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: Proc. TACAS, pp. 331–346. Springer (2003). [https://doi.org/10.1007/3-540-36577-X\\_24](https://doi.org/10.1007/3-540-36577-X_24)
71. Codish, M., Mulkers, A., Bruynooghe, M., de la Banda, M.G., Hermenegildo, M.: Improving abstract interpretations by combining domains. In: Proc. PEPM, pp. 194–205. ACM (1993). <https://doi.org/10.1145/154630.154650>
72. Cohen, H., Whaley, J., Wildt, J., Gorogiannis, N.: BuDDy: A BDD package. <http://sourceforge.net/p/buddy/>
73. Cordeiro, L.C., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. IEEE Trans. Software Eng. **38**(4), 957–974 (2012). <https://doi.org/10.1109/TSE.2011.59>
74. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In: Proc. POPL, pp. 238–252. ACM (1977). <https://doi.org/10.1145/512950.512973>
75. Cousot, P., Cousot, R.: Systematic design of program-analysis frameworks. In: Proc. POPL, pp. 269–282. ACM (1979). <https://doi.org/10.1145/567752.567778>
76. Cruanes, S., Hamon, G., Owre, S., Shankar, N.: Tool integration with the Evidential Tool Bus. In: Proc. VMCAI, LNCS 7737, pp. 275–294. Springer (2013). [https://doi.org/10.1007/978-3-642-35873-9\\_18](https://doi.org/10.1007/978-3-642-35873-9_18)
77. Cruanes, S., Heymans, S., Mason, I., Owre, S., Shankar, N.: The semantics of Datalog for the Evidential Tool Bus. In: Specification, Algebra, and Software, pp. 256–275. Springer (2014). [https://doi.org/10.1007/978-3-642-54624-2\\_13](https://doi.org/10.1007/978-3-642-54624-2_13)
78. 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>
79. Csallner, C., Smaragdakis, Y., Xie, T.: DSD-Crasher: A hybrid analysis tool for bug finding. ACM Transactions on Software Engineering and Methodology **17**(2), 1–37 (2008). <https://doi.org/10.1145/1348250.1348254>
80. Czech, M., Jakobs, M., Wehrheim, H.: Just test what you cannot verify! In: Proc. FASE, LNCS 9033, pp. 100–114. Springer (2015). [https://doi.org/10.1007/978-3-662-46675-9\\_7](https://doi.org/10.1007/978-3-662-46675-9_7)
81. Daca, P., Gupta, A., Henzinger, T.A.: Abstraction-driven concolic testing. In: Proc. VMCAI, LNCS 9583, pp. 328–347. Springer (2016). [https://doi.org/10.1007/978-3-662-49122-5\\_16](https://doi.org/10.1007/978-3-662-49122-5_16)
82. Dan, A., Meshman, Y., Vechev, M., Yahav, E.: Effective Abstractions for Verification under Relaxed Memory Models. In: Proc. VMCAI, pp. 449–466. Springer (2015). [https://doi.org/10.1007/978-3-662-46081-8\\_25](https://doi.org/10.1007/978-3-662-46081-8_25)
83. David, C., Kesseli, P., Kroening, D., Lewis, M.: Danger Invariants. In: Proc. FM, pp. 182–198. Springer (2016). [https://doi.org/10.1007/978-3-319-48989-6\\_12](https://doi.org/10.1007/978-3-319-48989-6_12)
84. Deng, D., Zhang, W., Lu, S.: Efficient concurrency-bug detection across inputs. In: Proc. OOPSLA. ACM (2013). <https://doi.org/10.1145/2509136.2509539>
85. Dhok, M., Ramanathan, M.K.: Directed test generation to detect loop inefficiencies. In: Proc. ESEC/FSE. ACM (2016). <https://doi.org/10.1145/2950290.2950360>
86. van Dijk, T.: Sylvan: Multi-core decision diagrams. Ph.D. thesis, University of Twente, Enschede, Netherlands (2016)
87. Dilling, I., Dilling, T., Li, B., McMillan, K.L.: Inductive invariant generation via abductive inference. In: Proc. OOPSLA, pp. 443–456. ACM (2013). <https://doi.org/10.1145/2509136.2509511>
88. Donaldson, A.F., Haller, L., Kröning, D., Rümmer, P.: Software verification using k-induction. In: Proc. SAS, LNCS 6887, pp. 351–368. Springer (2011). [https://doi.org/10.1007/978-3-642-23702-7\\_26](https://doi.org/10.1007/978-3-642-23702-7_26)
89. D’Silva, V., Kröning, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. on CAD of Integrated Circuits and Systems **27**(7), 1165–1178 (2008). <https://doi.org/10.1109/TCAD.2008.923410>
90. Elkader, K.A., Grumberg, O., Păsăreanu, C.S., Shoham, S.: Automated circular assume-guarantee reasoning. In: Proc. FM, pp. 23–39. Springer (2015). [https://doi.org/10.1007/978-3-319-19249-9\\_3](https://doi.org/10.1007/978-3-319-19249-9_3)
91. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Trans. Softw. Eng. **27**(2), 1–25 (2001). <https://doi.org/10.1109/32.908957>
92. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The DAIKON system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007). <https://doi.org/10.1016/j.scico.2007.01.015>
93. Farzan, A., Madhusudan, P., Razavi, N., Sorrentino, F.: Predicting null-pointer dereferences in concurrent programs. In: Proc. FSE. ACM (2012). <https://doi.org/10.1145/2393596.2393651>
94. Fedyukovich, G., D’Iddio, A.C., Hyvärinen, A.E.J., Sharygina, N.: Symbolic Detection of Assertion Dependencies for Bounded Model Checking. In: Proc. FASE, pp. 186–201. Springer (2015). [https://doi.org/10.1007/978-3-662-46675-9\\_13](https://doi.org/10.1007/978-3-662-46675-9_13)
95. Ferles, K., Wüstholtz, V., Christakis, M., Dilling, I.: Failure-directed program trimming. In: Proc. ESEC/FSE, pp. 174–185. ACM (2017). <https://doi.org/10.1145/3106237.3106249>
96. Fischer, B., Inverso, O., Parlato, G.: CSeq: A concurrency pre-processor for sequential C verification tools. In: Proc. ASE, pp. 710–713. IEEE (2013). <https://doi.org/10.1109/ASE.2013.6693139>
97. Fischer, J., Jhala, R., Majumdar, R.: Joining data flow with predicates. In: Proc. FSE, pp. 227–236. ACM (2005). <https://doi.org/10.1145/1081706.1081742>
98. Flanagan, C., Joshi, R., Leino, K.R.M.: Annotation inference for modular checkers. Information Processing Letters (to appear) (2001). [https://doi.org/10.1016/S0020-0190\(00\)00196-4](https://doi.org/10.1016/S0020-0190(00)00196-4)
99. Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM (2011). <https://doi.org/10.1145/2025113.2025179>
100. Fraser, G., Wotawa, F., Ammann, P.: Testing with model checkers: A survey. STVR **19**(3), 215–261 (2009). <https://doi.org/10.1002/stvr.402>
101. Fulton, N., Mitsch, S., Quesel, J.D., Völpl, M., Platzer, A.: Keymaera x: An axiomatic tactical theorem prover for hybrid systems. In: Automated Deduction - CADE-25, pp. 527–538. Springer International Publishing (2015). [https://doi.org/10.1007/978-3-319-21401-6\\_36](https://doi.org/10.1007/978-3-319-21401-6_36)
102. Ganai, M.K.: Efficient data race prediction with incremental reasoning on time-stamped lock history. In: Proc. ASE. IEEE (2013). <https://doi.org/10.1109/ase.2013.6693064>
103. Garg, P., Ivancic, F., Balakrishnan, G., Maeda, N., Gupta, A.: Feedback-directed unit test generation for C/C++ using concolic execution. In: Proc. ICSE. IEEE (2013). <https://doi.org/10.1109/icse.2013.6606559>

104. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Proc. POPL. ACM (2016). <https://doi.org/10.1145/2837614.2837664>
105. Gario, M., Micheli, A.: PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In: Proc. SMT (2015)
106. Ge, X., Taneja, K., Xie, T., Tillmann, N.: DyTa: Dynamic symbolic execution guided with static verification results. In: Proc. ICSE, pp. 992–994. ACM (2011). <https://doi.org/10.1145/1985793.1985971>
107. Gheorghiu Bobaru, M., Păsăreanu, C.S., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In: Proc. CAV, pp. 135–148. Springer (2008). [https://doi.org/10.1007/978-3-540-70545-1\\_14](https://doi.org/10.1007/978-3-540-70545-1_14)
108. Giacobbe, M., Kroening, D., Parsert, J.: Neural termination analysis. In: Proc. ESEC/FSE, pp. 633–645. ACM (2022). <https://doi.org/10.1145/3540250.3549120>
109. Giunchiglia, E., Narizzano, M., Tacchella, A., Vardi, M.Y.: Towards an efficient library for SAT: A manifesto. Electron. Notes Discret. Math. **9**, 290–310 (2001). [https://doi.org/10.1016/S1571-0653\(04\)00329-4](https://doi.org/10.1016/S1571-0653(04)00329-4)
110. Godefroid, P.: Model checking for programming languages using VeriSoft. In: Proc. POPL, pp. 174–186. ACM (1997). <https://doi.org/10.1145/263699.263717>
111. Godefroid, P.: Compositional dynamic test generation. In: Proc. POPL. ACM (2007). <https://doi.org/10.1145/1190216.1190226>
112. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proc. PLDI, pp. 213–223. ACM (2005). <https://doi.org/10.1145/1065010.1065036>
113. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: Unleashing the power of alternation. In: Proc. POPL, pp. 43–56. ACM (2010). <https://doi.org/10.1145/1706299.1706307>
114. Google: american fuzzy lop. <https://github.com/google/AFL>. Accessed: 2023-03-08
115. Gotlieb, A., Marijan, D.: FLOWER: optimal test suite reduction as a network maximum flow. In: Proc. ISSTA. ACM (2014). <https://doi.org/10.1145/2610384.2610416>
116. Graf, S., Saïdi, H.: Construction of abstract state graphs with Pvs. In: Proc. CAV, LNCS 1254, pp. 72–83. Springer (1997). [https://doi.org/10.1007/3-540-63166-6\\_10](https://doi.org/10.1007/3-540-63166-6_10)
117. Gropp, W.D., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message-passing interface, vol. 1. MIT press (1999). ISBN: 026257134X
118. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: A new algorithm for property checking. In: Proc. FSE, pp. 117–127. ACM (2006). <https://doi.org/10.1145/1181775.1181790>
119. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: Proc. PLDI, pp. 376–386. ACM (2006). <https://doi.org/10.1145/1133981.1134026>
120. Guo, S., Kusano, M., Wang, C., Yang, Z., Gupta, A.: Assertion guided symbolic execution of multithreaded programs. In: Proc. ESEC/FSE, p. 854–865. ACM (2015). <https://doi.org/10.1145/2786805.2786841>
121. Gupta, A., Henzinger, T.A., Radhakrishna, A., Samanta, R., Tarrach, T.: Succinct Representation of Concurrent Trace Sets. In: Proc. POPL. ACM (2015). <https://doi.org/10.1145/2676726.2677008>
122. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Proc. CAV, LNCS 9206, pp. 343–361. Springer (2015). [https://doi.org/10.1007/978-3-319-21690-4\\_20](https://doi.org/10.1007/978-3-319-21690-4_20)
123. Haltermann, J., Wehrheim, H.: CoVEGI: Cooperative verification via externally generated invariants. In: Proc. FASE, LNCS 12649, pp. 108–129 (2021). [https://doi.org/10.1007/978-3-030-71500-7\\_6](https://doi.org/10.1007/978-3-030-71500-7_6)
124. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Proc. SAS, LNCS 5673, pp. 69–85. Springer (2009). [https://doi.org/10.1007/978-3-642-03237-0\\_7](https://doi.org/10.1007/978-3-642-03237-0_7)
125. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV, LNCS 8044, pp. 36–52. Springer (2013). [https://doi.org/10.1007/978-3-642-39799-8\\_2](https://doi.org/10.1007/978-3-642-39799-8_2)
126. Helm, D., Kübler, F., Reif, M., Eichberg, M., Mezini, M.: Modular collaborative program analysis in OPAL. In: Proc. ESEC/FSE. ACM (2020). <https://doi.org/10.1145/3368089.3409765>
127. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL, pp. 58–70. ACM (2002). <https://doi.org/10.1145/503272.503279>
128. Herman, P.M.: A data flow analysis approach to program testing. Australian Computer Journal **8**(3), 92–96 (1976)
129. Hermenegildo, M.V.: Conferences vs. Journals in CS, what to do? Evolutionary ways forward and the ICLP/PLP model. In: Dagstuhl 12452: Publication Culture in Computing Research - Position Papers, vol. 12452 (2012)
130. Hoare, C.A.R.: Procedures and parameters: An axiomatic approach. In: Symposium on Semantics of Algorithmic Languages, pp. 102–116. Springer (1971). <https://doi.org/10.1007/BFb0059696>
131. Hong, S., Ahn, J., Park, S., Kim, M., Harrold, M.J.: Testing concurrent programs to achieve high synchronization coverage. In: Proc. ISSTA. ACM (2012). <https://doi.org/10.1145/2338965.2336779>
132. Hu, Y., Neamtii, I., Alavi, A.: Automatically verifying and reproducing event-based races in Android apps. In: Proc. ISSTA. ACM (2016). <https://doi.org/10.1145/2931037.2931069>
133. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science **275**(7), 51–54 (1997). <https://doi.org/10.1126/science.275.5296.51>
134. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded model checking of multi-threaded C programs via lazy sequentialization. In: Proc. CAV, LNCS 8559, pp. 585–602. Springer (2014). [https://doi.org/10.1007/978-3-319-08867-9\\_39](https://doi.org/10.1007/978-3-319-08867-9_39)
135. Jakobs, M.: PART  $\backslash$   $\mathrm{PW}$ : From partial analysis results to a proof witness. In: Proc. SEFM, LNCS, vol. 10469, pp. 120–135. Springer (2017). [https://doi.org/10.1007/978-3-319-66197-1\\_8](https://doi.org/10.1007/978-3-319-66197-1_8)
136. Jakobs, M., Wiesner, M.: PEQtest: Testing Functional Equivalence. In: Proc. FASE, pp. 184–204. Springer (2022). [https://doi.org/10.1007/978-3-030-99429-7\\_11](https://doi.org/10.1007/978-3-030-99429-7_11)
137. Jensen, C.S., Prasad, M.R., Möller, A.: Automated testing with targeted event sequence generation. In: Proc. ISSTA. ACM (2013). <https://doi.org/10.1145/2483760.2483777>
138. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41**(4) (2009). <https://doi.org/10.1145/1592434.1592438>
139. Jones, C.B.: Tentative steps toward a development method for interfering programs. TOPLAS **5**(4), 596–619 (1983). <https://doi.org/10.1145/69575.69577>
140. Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: A survey. J. Artif. Intell. Res. **4**, 237–285 (1996). <https://doi.org/10.1613/jair.301>
141. Kaleeswaran, A.P., Nordmann, A., Vogel, T., Grunske, L.: A systematic literature review on counterexample explanation. Information and Software Technology **145**, 106800 (2022). <https://doi.org/10.1016/j.infsof.2021.106800>

142. Karna, A.K., Chen, Y., Yu, H., Zhong, H., Zhao, J.: The role of model checking in software engineering. *Frontiers Comput. Sci.* **12**(4), 642–668 (2018). <https://doi.org/10.1007/s11704-016-6192-0>
143. Kharkar, A., Moghaddam, R.Z., Jin, M., Liu, X., Shi, X., Clement, C., Sundaresan, N.: Learning to reduce false positives in analytic bug detectors. In: *Proc. ICSE*. ACM (2022). <https://doi.org/10.1145/3510003.3510153>
144. King, J.C.: Symbolic execution and program testing. *Commun. ACM* **19**(7), 385–394 (1976). <https://doi.org/10.1145/360248.360252>
145. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. *Formal Aspects Comput.* **27**(3), 573–609 (2015). <https://doi.org/10.1007/s00165-014-0326-7>
146. Koroglu, Y., Sen, A.: TCM: Test Case Mutation to Improve Crash Detection in Android. In: *Proc. FASE*, pp. 264–280. Springer (2018). [https://doi.org/10.1007/978-3-319-89363-1\\_15](https://doi.org/10.1007/978-3-319-89363-1_15)
147. Koroglu, Y., Sen, A., Muslu, O., Mete, Y., Ulker, C., Tanriverdi, T., Donmez, Y.: Qbe: Qlearning-based exploration of android applications. In: *Proc. ICST*. IEEE (2018). <https://doi.org/10.1109/icst.2018.00020>
148. Kotoun, M., Peringer, P., Šoková, V., Vojnar, T.: Optimized PREDATORHP and the SV-COMP heap and memory safety benchmark (competition contribution). In: *Proc. TACAS, LNCS 9636*, pp. 942–945. Springer (2016). [https://doi.org/10.1007/978-3-662-49674-9\\_66](https://doi.org/10.1007/978-3-662-49674-9_66)
149. Kroening, D., Lewis, M., Weissenbacher, G.: Under-Approximating Loops in C Programs for Fast Counterexample Detection. In: *Proc. CAV*, pp. 381–396. Springer (2013). [https://doi.org/10.1007/978-3-642-39799-8\\_26](https://doi.org/10.1007/978-3-642-39799-8_26)
150. Kroening, D., Lewis, M., Weissenbacher, G.: Proving Safety with Trace Automata and Bounded Model Checking. In: *Proc. FM*, pp. 325–341. Springer (2015). [https://doi.org/10.1007/978-3-319-19249-9\\_21](https://doi.org/10.1007/978-3-319-19249-9_21)
151. Kukucka, J., Pina, L., Ammann, P., Bell, J.: CONFETTI. In: *Proc. ICSE*. ACM (2022). <https://doi.org/10.1145/3510003.3510628>
152. Kumar, S., Sanyal, A., Venkatesh, R., Shah, P.: Property Checking Array Programs Using Loop Shrinking. In: *Proc. TACAS*, pp. 213–231. Springer (2018). [https://doi.org/10.1007/978-3-319-89960-2\\_12](https://doi.org/10.1007/978-3-319-89960-2_12)
153. Kusano, M., Wang, C.: Assertion guided abstraction. In: *Proc. ASE*. ACM (2014). <https://doi.org/10.1145/2642937.2642998>
154. Lahiri, S., Roy, S.: Almost correct invariants: synthesizing inductive invariants by fuzzing proofs. In: *Proc. ISSTA*. ACM (2022). <https://doi.org/10.1145/3533767.3534381>
155. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: *Proc. FSE*, pp. 345–355. ACM (2013). <https://doi.org/10.1145/2491411.2491452>
156. Lampropoulos, L., Hicks, M., Pierce, B.C.: Coverage guided, property based testing. *PACMPL* **3**(OOPSLA), 1–29 (2019). <https://doi.org/10.1145/3360607>
157. Lampson, B.W.: Software Components: Only the Giants Survive, pp. 137–145. Springer (2004). [https://doi.org/10.1007/0-387-21821-1\\_21](https://doi.org/10.1007/0-387-21821-1_21)
158. Lanzinger, F., Weigl, A., Ulbrich, M., Dietl, W.: Scalability and precision by combining expressive type systems and deductive verification. *PACMPL* **5**(OOPSLA), 1–29 (2021). <https://doi.org/10.1145/3485520>
159. Le-Cong, T., Kang, H.J., Nguyen, T.G., Haryono, S.A., Lo, D., Le, X.D., Huynh, Q.T.: Autopruner: Transformer-based call graph pruning. In: *Proc. ESEC/FSE*, p. 520–532. ACM (2022). <https://doi.org/10.1145/3540250.3549175>
160. Lee, S., Lee, H., Ryu, S.: Broadening horizons of multilingual static analysis. In: *Proc. ASE*. ACM (2020). <https://doi.org/10.1145/3324884.3416558>
161. Lee, W., Lee, W., Yi, K.: Sound Non-statistical Clustering of Static Analysis Alarms. In: *Proc. VMCAI*, pp. 299–314. Springer (2012). [https://doi.org/10.1007/978-3-642-27940-9\\_20](https://doi.org/10.1007/978-3-642-27940-9_20)
162. Lerner, S., Grove, D., Chambers, C.: Composing data-flow analyses and transformations. In: *Proc. POPL*, pp. 270–282. ACM (2002). <https://doi.org/10.1145/503272.503298>
163. Li, C., Chen, R., Wang, B., Yu, T., Gao, D., Yang, M.: Precise and efficient atomicity violation detection for interrupt-driven programs via staged path pruning. In: *Proc. ISSTA*. ACM (2022). <https://doi.org/10.1145/3533767.3534412>
164. Li, J., Sun, J., Li, L., Le, Q.L., Lin, S.W.: Automatic loop-invariant generation and refinement through selective sampling. In: *Proc. ASE*, pp. 782–792. IEEE (2017). <https://doi.org/10.1109/ASE.2017.8115689>
165. Li, K., Reichenbach, C., Csallner, C., Smaragdakis, Y.: Residual investigation: predictive and precise bug detection. In: *Proc. ISSTA*, pp. 298–308. ACM (2012). <https://doi.org/10.1145/2338965.2336789>
166. Li, L., Bissyandé, T.F., Octeau, D., Klein, J.: DroidRA: taming reflection to support whole-program analysis of Android apps. In: *Proc. ISSTA*. ACM (2016). <https://doi.org/10.1145/2931037.2931044>
167. Li, M., Chen, Y., Wang, L., Xu, G.: Dynamically validating static memory leak warnings. In: *Proc. ISSTA*. ACM (2013). <https://doi.org/10.1145/2483760.2483778>
168. Lin, S., Sun, J., Nguyen, T.K., Liu, Y., Dong, J.S.: Interpolation Guided Compositional Verification (T). In: *Proc. ASE*. IEEE (2015). <https://doi.org/10.1109/ase.2015.33>
169. Liu, J., Wu, D., Xue, J.: TDroid: exposing app switching attacks in Android with control flow specialization. In: *Proc. ASE*. ACM (2018). <https://doi.org/10.1145/3238147.3238188>
170. Lohar, D., Jeangoudoux, C., Sobel, J., Darulova, E., Christakis, M.: A Two-Phase Approach for Conditional Floating-Point Verification. In: *Proc. TACAS*, pp. 43–63. Springer (2021). [https://doi.org/10.1007/978-3-030-72013-1\\_3](https://doi.org/10.1007/978-3-030-72013-1_3)
171. Ma, L., Artho, C., Zhang, C., Sato, H., Gmeiner, J., Ramler, R.: GRT: Program-Analysis-Guided Random Testing (T). In: *Proc. ASE*. IEEE (2015). <https://doi.org/10.1109/ase.2015.49>
172. Majumdar, R., Sen, K.: Hybrid concolic testing. In: *Proc. ICSE*, pp. 416–426. IEEE (2007). <https://doi.org/10.1109/ICSE.2007.41>
173. Manès, V.J.M., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Software Eng.* **47**(11), 2312–2331 (2021). <https://doi.org/10.1109/TSE.2019.2946563>
174. Mann, M., Wilson, A., Zohar, Y., Stuntz, L., Irfan, A., Brown, K., Donovan, C., Guman, A., Tinelli, C., Barrett, C.: Smt-switch: a solver-agnostic c++ api for smt solving. In: *Proc. SAT*, pp. 377–386. Springer (2021). [https://doi.org/10.1007/978-3-030-80223-3\\_26](https://doi.org/10.1007/978-3-030-80223-3_26)
175. Margaria, T., Braun, V., Kreidler, J.: Interacting with ETI: A user session. *STTT* **1**(1–2), 49–63 (1997). <https://doi.org/10.1007/s100090050005>
176. Margaria, T., Nagel, R., Steffen, B.: Remote integration and coordination of verification tools in jETI. In: *Proc. ECBS*, pp. 431–436 (2005). <https://doi.org/10.1109/ECBS.2005.59>
177. Margaria, T., Nagel, R., Steffen, B.: jETI: A tool for remote tool integration. In: *Proc. TACAS, LNCS 3440*, pp. 557–562. Springer (2005). [https://doi.org/10.1007/978-3-540-31980-1\\_38](https://doi.org/10.1007/978-3-540-31980-1_38)




178. Margaria, T., Steffen, B.: LTL guided planning: Revisiting automatic tool composition in ETI. In: Proc. SEW, pp. 214–226. IEEE (2007). <https://doi.org/10.1109/SEW.2007.101>
179. McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. CAV, LNCS 4144, pp. 123–136. Springer (2006). [https://doi.org/10.1007/11817963\\_14](https://doi.org/10.1007/11817963_14)
180. Meng, R., Dong, Z., Li, J., Beschastnikh, I., Roychoudhury, A.: Linear-time temporal logic guided greybox fuzzing. In: Proc. ICSE. ACM (2022). <https://doi.org/10.1145/3510003.3510082>
181. Molina, F., d’Amorim, M., Aguirre, N.: Fuzzing class specifications. In: Proc. ICSE. ACM (2022). <https://doi.org/10.1145/3510003.3510120>
182. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS, LNCS 4963, pp. 337–340. Springer (2008). [https://doi.org/10.1007/978-3-540-78800-3\\_24](https://doi.org/10.1007/978-3-540-78800-3_24)
183. Mukherjee, R., Schrammel, P., Haller, L., Kroening, D., Melham, T.: Lifting CDCL to Template-Based Abstract Domains for Program Verification. In: Proc. ATVA, pp. 307–326. Springer (2017). [https://doi.org/10.1007/978-3-319-68167-2\\_21](https://doi.org/10.1007/978-3-319-68167-2_21)
184. Naik, M., Yang, H., Castelnovo, G., Sagiv, M.: Abstractions from tests. In: Proc. POPL. ACM (2012). <https://doi.org/10.1145/2103656.2103701>
185. Neider, D., Garg, P., Madhusudan, P., Saha, S., Park, D.: Invariant Synthesis for Incomplete Verification Engines. In: Proc. TACAS, pp. 232–250. Springer (2018). [https://doi.org/10.1007/978-3-319-89960-2\\_13](https://doi.org/10.1007/978-3-319-89960-2_13)
186. Nguyen, T., Antonopoulos, T., Ruef, A., Hicks, M.: Counterexample-guided approach to finding numerical invariants. In: Proc. ESEC/FSE, p. 605–615. ACM (2017). <https://doi.org/10.1145/3106237.3106281>
187. Nguyen, T., Dwyer, M.B., Visser, W.: SymInfer: Inferring program invariants using symbolic states. In: Proc ASE, pp. 804–814 (2017). <https://doi.org/10.1109/ASE.2017.8115691>
188. Nguyen, T., Kapur, D., Weimer, W., Forrest, S.: Dig: A dynamic invariant generator for polynomial and array invariants. ACM Trans. Softw. Eng. Methodol. **23**(4) (2014). <https://doi.org/10.1145/2556782>
189. Nguyen, T.L., Schrammel, P., Fischer, B., La Torre, S., Parlato, G.: Parallel bug-finding in concurrent programs via reduced interleaving instances. In: Proc. ASE, pp. 753–764. IEEE (2017). <https://doi.org/10.1109/ASE.2017.8115686>
190. 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>
191. Noller, Y., Kersten, R., Pasareanu, C.S.: Badger: Complexity analysis with fuzzing and symbolic execution. In: Proc. ISSTA, pp. 322–332. ACM (2018). <https://doi.org/10.1145/3213846.3213868>
192. Nori, A.V., Sharma, R.: Termination proofs from tests. In: Proc. ESEC/FSE. ACM (2013). <https://doi.org/10.1145/2491411.2491413>
193. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: Proc. ICSE, pp. 75–84. IEEE (2007). <https://doi.org/10.1109/ICSE.2007.37>
194. Padhi, S., Sharma, R., Millstein, T.: Data-driven precondition inference with learned features. In: Proc. PLDI. ACM (2016). <https://doi.org/10.1145/2908080.2908099>
195. Pastore, F., Mariani, L., Hyvärinen, A.E.J., Fedyukovich, G., Sharygina, N., Sehested, S., Muhammad, A.: Verification-aided regression testing. In: Proc. ISSTA. ACM (2014). <https://doi.org/10.1145/2610384.2610387>
196. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Logics and Models of Concurrent Systems, pp. 123–144. Springer (1985). [https://doi.org/10.1007/978-3-642-82453-1\\_5](https://doi.org/10.1007/978-3-642-82453-1_5)
197. Post, H., Sinz, C., Kaiser, A., Gorges, T.: Reducing false positives by combining abstract interpretation and bounded model checking. In: Proc. ASE, pp. 188–197. IEEE (2008). <https://doi.org/10.1109/ASE.2008.29>
198. Pradel, M., Gross, T.R.: Fully automatic and precise detection of thread safety violations. In: Proc. PLDI. ACM (2012). <https://doi.org/10.1145/2254064.2254126>
199. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al.: Language models are unsupervised multitask learners. OpenAI blog **1**(8), 9 (2019)
200. Rapps, S., Weyuker, E.: Selecting software test data using data flow information. IEEE Transactions on Software Engineering **SE-11**(4), 367–375 (1985). <https://doi.org/10.1109/tse.1985.232226>
201. Rasin, D., Grumberg, O., Shoham, S.: Modular verification of concurrent programs via sequential model checking. In: Proc. ATVA, pp. 228–247. Springer (2018). [https://doi.org/10.1007/978-3-030-01090-4\\_14](https://doi.org/10.1007/978-3-030-01090-4_14)
202. Recoules, F., Bardin, S., Bonichon, R., Mounier, L., Potet, M.: Get Rid of Inline Assembly through Verification-Oriented Lifting. In: Proc. ASE. IEEE (2019). <https://doi.org/10.1109/ase.2019.00060>
203. Rice, J.R.: The algorithm selection problem. Advances in Computers **15**, 65–118 (1976). [https://doi.org/10.1016/S0065-2458\(08\)60520-3](https://doi.org/10.1016/S0065-2458(08)60520-3)
204. Richter, C., Wehrheim, H.: Attend and represent: A novel view on algorithm selection for software verification. In: Proc. ASE, pp. 1016–1028 (2020). <https://doi.org/10.1145/3324884.3416633>
205. Riley, D., Fedyukovich, G.: Multi-phase invariant synthesis. In: Proc. ESEC/FSE, p. 607–619. ACM (2022). <https://doi.org/10.1145/3540250.3549166>
206. Rocha, H.O., Barreto, R.S., Cordeiro, L.C.: Memory management test-case generation of C programs using bounded model checking. In: Proc. SEFM, LNCS 9276, pp. 251–267. Springer (2015). [https://doi.org/10.1007/978-3-319-22969-0\\_18](https://doi.org/10.1007/978-3-319-22969-0_18)
207. Roemer, J., Genç, K., Bond, M.D.: High-coverage, unbounded sound predictive race detection. In: Proc. PLDI. ACM (2018). <https://doi.org/10.1145/3192366.3192385>
208. Rushby, J.M.: An Evidential Tool Bus. In: Proc. ICFEM, LNCS 3785, pp. 36–36. Springer (2005). [https://doi.org/10.1007/11576280\\_3](https://doi.org/10.1007/11576280_3)
209. Salehnamadi, N., Alshayban, A., Ahmed, I., Malek, S.: ER catcher. In: Proc. ASE. ACM (2020). <https://doi.org/10.1145/3324884.3416639>
210. Samak, M., Ramanathan, M.K., Jagannathan, S.: Synthesizing racy tests. In: Proc. PLDI. ACM (2015). <https://doi.org/10.1145/2737924.2737998>
211. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: Proc. FSE, pp. 263–272. ACM (2005). <https://doi.org/10.1145/1081706.1081750>
212. Sery, O., Fedyukovich, G., Sharygina, N.: Funfrog: Bounded model checking with interpolation-based function summarization. In: Proc. ATVA, LNCS 7561, pp. 203–207. Springer (2012). [https://doi.org/10.1007/978-3-642-33386-6\\_17](https://doi.org/10.1007/978-3-642-33386-6_17)
213. Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-based function summaries in bounded model checking. In: Proc. HVC, LNCS 7261, pp. 160–175. Springer (2012). [https://doi.org/10.1007/978-3-642-34188-5\\_15](https://doi.org/10.1007/978-3-642-34188-5_15)
214. Shan, Z., Azim, T., Neamtiu, I.: Finding resume and restart errors in Android applications. In: Proc. OOPSLA. ACM (2016). <https://doi.org/10.1145/2983990.2984011>



215. Shankar, N.: Little engines of proof. In: Proc. FME, LNCS 2391, pp. 1–20. Springer (2002). [https://doi.org/10.1007/3-540-45614-7\\_1](https://doi.org/10.1007/3-540-45614-7_1)
216. Sherman, E., Dwyer, M.B.: Structurally defined conditional data-flow static analysis. In: Proc. TACAS (2), LNCS 10806, pp. 249–265. Springer (2018). [https://doi.org/10.1007/978-3-319-89963-3\\_15](https://doi.org/10.1007/978-3-319-89963-3_15)
217. Simic, S., Inverso, O., Tribastone, M.: Bit-precise verification of discontinuity errors under fixed-point arithmetic. In: Proc. SEFM, LNCS, vol. 13085, pp. 443–460. Springer (2021). [https://doi.org/10.1007/978-3-030-92124-8\\_25](https://doi.org/10.1007/978-3-030-92124-8_25)
218. Sogokon, A., Mitsch, S., Tan, Y.K., Cordwell, K., Platzer, A.: Pegasus: A Framework for Sound Continuous Invariant Generation. In: Proc. FM, pp. 138–157. Springer (2019). [https://doi.org/10.1007/978-3-030-30942-8\\_10](https://doi.org/10.1007/978-3-030-30942-8_10)
219. Somenzi, F.: Colorado University decision diagram package (1998)
220. Sondhi, D., Purandare, R.: SEGATE: Unveiling Semantic Inconsistencies between Code and Specification of String Inputs. In: Proc. ASE. IEEE (2019). <https://doi.org/10.1109/ase.2019.00028>
221. Spector, A.Z.: Modular architectures for distributed and database systems. In: Proc. PODS, pp. 217–224. ACM (1989). <https://doi.org/10.1145/73721.73743>
222. Steffen, B.: The physics of software tools: SWOT analysis and vision. Int. J. Softw. Tools Technol. Transf. **19**(1), 1–7 (2017). <https://doi.org/10.1007/s10009-016-0446-x>
223. Steffen, B., Margaria, T., Braun, V.: The Electronic Tool Integration platform: Concepts and design. STTT **1**(1–2), 9–30 (1997). <https://doi.org/10.1007/s1000900050003>
224. Su, T., Fu, Z., Pu, G., He, J., Su, Z.: Combining Symbolic Execution and Model Checking for Data Flow Testing. In: Proc. ICSE. IEEE (2015). <https://doi.org/10.1109/icse.2015.81>
225. Svyatkovskiy, A., Deng, S.K., Fu, S., Sundaresan, N.: Intellicode compose: code generation using transformer. In: Proc. ESEC/FSE. ACM (2020). <https://doi.org/10.1145/3368089.3417058>
226. Terragni, V., Cheung, S.: Coverage-driven test code generation for concurrent classes. In: Proc. ICSE. ACM (2016). <https://doi.org/10.1145/2884781.2884876>
227. Terragni, V., Cheung, S., Zhang, C.: RECONTEST: Effective Regression Testing of Concurrent Programs. In: Proc. ICSE. IEEE (2015). <https://doi.org/10.1109/icse.2015.45>
228. Turing, A.: On computable numbers, with an application to the Entscheidungsproblem. In: Proc. LMS, vol. s2-42, pp. 230–265. London Mathematical Society (1937). <https://doi.org/10.1112/plms/s2-42.1.230>
229. Utture, A., Liu, S., Kalhauge, C.G., Palsberg, J.: Striking a balance. In: Proc. ICSE. ACM (2022). <https://doi.org/10.1145/3510003.3510166>
230. Utture, A., Palsberg, J.: Fast and precise application code analysis using a partial library. In: Proc. ICSE. ACM (2022). <https://doi.org/10.1145/3510003.3510046>
231. Vardi, M.Y.: Conferences vs. journals in computing research. Commun. ACM **52**(5), 5 (2009). <https://doi.org/10.1145/1506409.1506410>
232. Visser, W., Geldenhuys, J., Dwyer, M.B.: GREEN: Reducing, reusing, and recycling constraints in program analysis. In: Proc. FSE, pp. 58:1–58:11. ACM (2012). <https://doi.org/10.1145/2393596.2393665>
233. Vizel, Y., Gurfinkel, A.: Interpolating Property Directed Reachability. In: Proc. CAV, pp. 260–276. Springer (2014). [https://doi.org/10.1007/978-3-319-08867-9\\_17](https://doi.org/10.1007/978-3-319-08867-9_17)
234. Vrettas, G., Sanderson, M.: Conferences versus journals in computer science. J. Assoc. Inf. Sci. Technol. **66**(12), 2674–2684 (2015). <https://doi.org/10.1002/asi.23349>
235. Wang, J., Wang, C.: Learning to synthesize relational invariants. In: Proc. ASE. ACM (2023). <https://doi.org/10.1145/3551349.3556942>
236. Wang, Y., Wang, L., Yu, T., Zhao, J., Li, X.: Automatic detection and validation of race conditions in interrupt-driven embedded software. In: Proc. ISSTA. ACM (2017). <https://doi.org/10.1145/3092703.3092724>
237. Wen, C., He, M., Wu, B., Xu, Z., Qin, S.: Controlled concurrency testing via periodical scheduling. In: Proc. ICSE. ACM (2022). <https://doi.org/10.1145/3510003.3510178>
238. Wendler, P.: CPAchecker with sequential combination of explicit-state analysis and predicate analysis (competition contribution). In: Proc. TACAS, LNCS 7795, pp. 613–615. Springer (2013). [https://doi.org/10.1007/978-3-642-36742-7\\_45](https://doi.org/10.1007/978-3-642-36742-7_45)
239. Winter, S., Timperley, C.S., Hermann, B., Cito, J., Bell, J., Hilton, M., Beyer, D.: A retrospective study of one decade of artifact evaluations. In: Proc. ESEC/FSE, pp. 145–156. ACM (2022). <https://doi.org/10.1145/3540250.3549172>
240. Wonisch, D., Wehrheim, H.: Predicate analysis with block-abstraction memoization. In: Proc. ICFEM, LNCS 7635, pp. 332–347. Springer (2012). [https://doi.org/10.1007/978-3-642-34281-3\\_24](https://doi.org/10.1007/978-3-642-34281-3_24)
241. Wüstholtz, V., Christakis, M.: Targeted greybox fuzzing with static lookahead analysis. In: Proc. ICSE. ACM (2020). <https://doi.org/10.1145/3377811.3380388>
242. Xu, R., Chen, J., He, F.: Data-driven loop bound learning for termination analysis. In: Proc. ICSE. ACM (2022). <https://doi.org/10.1145/3510003.3510220>
243. Yin, L., Dong, W., Liu, W., Wang, J.: Parallel refinement for multi-threaded program verification. In: Proc. ICSE, pp. 643–653. IEEE (2019). <https://doi.org/10.1109/ICSE.2019.00074>
244. Yu, H., Chen, Z., Fu, X., Wang, J., Su, Z., Sun, J., Huang, C., Dong, W.: Symbolic verification of message passing interface programs. In: Proc. ICSE. ACM (2020). <https://doi.org/10.1145/3377811.3380419>
245. Yu, T., Srisa-an, W., Rothermel, G.: SimRacer: an automated framework to support testing for process-level races. In: Proc. ISSTA. ACM (2013). <https://doi.org/10.1145/2483760.2483771>
246. Zeng, R., Sun, Z., Liu, S., He, X.: A method for improving the precision and coverage of atomicity violation predictions. In: Proc. TACAS, pp. 116–130. Springer (2015). [https://doi.org/10.1007/978-3-662-46681-0\\_8](https://doi.org/10.1007/978-3-662-46681-0_8)
247. Zhang, C., Groce, A., Alipour, M.A.: Using test case reduction and prioritization to improve symbolic execution. In: Proc. ISSTA. ACM (2014). <https://doi.org/10.1145/2610384.2610392>
248. Zhang, L., Yang, G., Rungta, N., Person, S., Khurshid, S.: Feedback-driven dynamic invariant discovery. In: Proc. ISSTA. ACM (2014). <https://doi.org/10.1145/2610384.2610389>
249. Zhu, H., Nori, A.V., Jagannathan, S.: Dependent Array Type Inference from Tests. In: Proc. VMCAI, pp. 412–430. Springer (2015). [https://doi.org/10.1007/978-3-662-46081-8\\_23](https://doi.org/10.1007/978-3-662-46081-8_23)
250. J. Švejda, Berger, P., Katoen, J.P.: Interpretation-based violation witness validation for C. NrvWit. In: Proc. TACAS, LNCS 12078, pp. 40–57. Springer (2020). [https://doi.org/10.1007/978-3-030-45190-5\\_3](https://doi.org/10.1007/978-3-030-45190-5_3)



# CoVeriTeam: On-Demand Composition of Cooperative Verification Systems

Dirk Beyer   and Sudeep Kanav 

LMU Munich, Munich, Germany

**Abstract.** There is no silver bullet for software verification: Different techniques have different strengths. Thus, it is imperative to combine the strengths of verification tools via combinations and cooperation. CoVeriTeam is a language and tool for on-demand composition of cooperative approaches. It provides a systematic and modular way to combine existing tools (without changing them) in order to leverage their full potential. The idea of cooperative verification is that different tools help each other to achieve the goal of correctly solving verification tasks.

The language is based on verification artifacts (programs, specifications, witnesses) as basic objects and verification actors (verifiers, validators, testers) as basic operations. We define composition operators that make it possible to easily describe new compositions. Verification artifacts are the interface between the different verification actors. CoVeriTeam consists of a language for composition of verification actors, and its interpreter.

As a result of viewing tools as components, we can now create powerful verification engines that are beyond the possibilities of single tools, avoiding to develop certain components repeatedly. We illustrate the abilities of CoVeriTeam on a few case studies. We expect that CoVeriTeam will help verification researchers and practitioners to easily experiment with new tools, and assist them in rapid prototyping of tool combinations.

**Keywords:** Cooperative Verification · Tool Development · Software Verification · Automatic Verification · Verification Tools · Tool Composition · Tool Reuse

## 1 Introduction

As research in the field of formal verification advanced, the complexity of the programs under verification also kept on increasing. As a result, despite its successful application to the source code of large industrial and open-source projects [2, 3, 23, 27, 36], the current techniques fall short on solving many important verification tasks. It seems essential to combine the strengths of different verification techniques and tools to solve these tasks.

The verification community successfully applies different approaches to combine ideas: integrated approaches (source-code-based), where different pieces of source code are integrated into one tool [28], and off-the-shelf approaches (executable-based), where different executables from existing tools are combined

2 Dirk Beyer and Sudeep Kanav

without changing them. The latter can be further classified into sequential and parallel portfolio [33], algorithm selection [37], and cooperative approaches [22].

The integrated approaches require development effort for adaptation or implementation of integrated components instead of building on existing mature implementations—the combination is very tight. On the other hand, the standard off-the-shelf approaches (portfolio [33] and selection [37]) let the tools run in isolation and the individual tools do not cooperate at all. The components do not benefit from the knowledge that is produced by other tools in the combination—the combination is very loose. In this work, we focus on cooperative verification, which is neither as tight as source-code integration nor as loose as portfolio and selection approaches—somewhere in between the two extremes.

Cooperative verification [22] is an approach to combine different tools for verification in such a way that they help each other solving a verification task, where the combinations are neither too tight nor too loose. Implementations include using a shared data base to exchange information (e.g., there are cooperative SAT solvers that use a shared set of learned clauses [34], and cooperative software verifiers that use a shared set of reached abstract states [14]) or pass information from one tool to the other (e.g., conditional model checkers [13, 25]). Cooperative verification aims to combine the individual strengths of these technologies to achieve better results. Our thesis is that *programming* (meta) verification systems based on *combination* and *cooperation* could be a promising solution. CoVeriTeam provides a framework to achieve this.

Developing such a tool is not straightforward. Various concerns that need to be addressed for developing a robust solution can be broadly divided in two categories: *concepts* and *execution*. (1) Concepts deal with defining the interfaces for tools, and with the mechanism for their combination. Before tools can cooperate, we need a common definition of tools based on their behavior. We need to categorize *what a tool does*, *what inputs it consumes*, and *what outputs it produces*, before we can use it in a cooperative framework with ease. In CoVeriTeam, we categorize tools in various types of verification *actors*, and the inputs and outputs produced by these actors in verification *artifacts*. The actors can be combined with the help of composition operators that define the mechanism of cooperation. (2) Execution is concerned with all issues during the execution of a tool. Actors first need to execute to cooperate. This opens another dimension of challenges and opportunities to improve the cooperation. To give two examples: a tool might have a too high resource consumption, thus, resources must be controlled and limited, and tools might interfere with other executing processes, thus, tools must be executed in isolated containers.

This paper presents CoVeriTeam, a language and tool for on-demand composition of cooperative verification systems that solves the above mentioned challenges. We contribute a domain-specific language and an execution engine. In the CoVeriTeam language, we can compose new actors based on existing ones using built-in composition operators. The existing components are not changed, but taken *off-the-shelf* from actor providers (technically: tool archives). We do not change existing software components: the composition is done *on-demand*

(when needed by the user) and *on-the-fly* (it does not compile a new tool from the components). In other words, existing verification tools are viewed as off-the-shelf components, and can be used in a larger context to construct more powerful verification compositions. Our approach does not require writing code in programming languages used to develop the underlying components. In the CoVeriTeam language, the user can execute tools without fearing that they interact with the host system or with other tools in an unspecified way. The execution environment, as well as input and output, are controlled using the Linux features cgroups, name spaces, and overlay file systems. We use the BENCHEXEC [20] system as library for convenient access to those OS features via a Python API.

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

1. a language to compose new verification tools based on existing ones,
2. an execution engine for on-the-fly execution of these compositions,
3. case studies implementing combinations in CoVeriTeam that were previously achieved only via hard-wired combinations, and
4. an open-source implementation and an artifact for reproduction.

In addition to the above mentioned contributions, CoVeriTeam provides the following features to the end user: (1) CoVeriTeam takes care of downloading and installing specified verification tools on the host system. (2) There is no need to learn command-line parameters of a verification tool because CoVeriTeam takes care of translating the input to the arguments for the underlying tool. This provides a uniform interface to a number of similar tools. (3) The off-the-shelf components (i.e., tools) are executed in a container, with resource limits, such that the execution cannot change (or even damage) the host system.

These features in turn enable a researcher or practitioner to easily experiment with new tools, and rapidly prototype new verification combinations. CoVeriTeam liberates the researcher who uses tool combinations from maintaining scripts that combine tools executions, and worrying about downloading, installing, and figuring out the command to execute a verification tool.

**Impact.** CoVeriTeam has already found use cases in the verification community: (1) It was used in a modular implementation of CEGAR [26] using off-the-shelf components [12]. (2) It was used for construction and evaluation of various verifier combinations [17]. (3) CoVeriTeam (wrapped in a service) was used in the software-verification competition 2021 and 2022 to help the participants debug issues with their tools (see Sect. 3 in [7]). Also, according to SV-COMP rules, a team is granted points only for those tasks whose result can be validated using a validator. Thus, a verifier-validator combination might be interesting for participants. With the help of CoVeriTeam such combinations can be easily constructed.

Also, the advent of many high-quality verifiers should lead to a certain level of standardization of the API and provided features. For example, tools for SMT or SAT solving are easy to use because of their standardized input language (e.g., SMTLIB for SMT solvers [4]). Consequently, such tools can be easily integrated into larger architectures as components. Our vision is that soon verifiers will be seen also as components that can be used in larger architectures just like SMT solvers are now integrated into verification tools.

4 Dirk Beyer and Sudeep Kanav

---

**Example 1** Witness Validation
 

---

**Input:** Program  $p$ , Specification  $s$

**Output:** Verdict

```

1: verifier := Verifier("Ultimate Automizer")
2: validator := Validator("CPAchecker")
3: result := verifier.verify(p, s)
4: if result.verdict  $\in$  {TRUE, FALSE} then
5:   result = validator.validate (p, s, result.witness)
6: return (result.verdict, result.witness)
  
```

---

## 2 Running Example

We explain the idea of CoVeriTeam using a short example. Verifiers are complex software systems and might have bugs. Therefore, for more assurance a user might want to validate the result of a verifier based on the verification witness that the verifier produces [10]. Such a procedure is sketched in [Example 1](#).

The user wanting to execute the procedure sketched in [Example 1](#) would need to download the tools (verifier and validator), execute the verifier, check the result of the verifier, and then if needed connect the outputs of the verifier with the inputs of the validator. The user would quite possibly write a shell script to do this, which is cumbersome and difficult to maintain.

CoVeriTeam takes care of all the above issues. In the next section, we discuss the types, namely artifacts and actors, that are used in the CoVeriTeam language. After this, we explain the design and usage of the CoVeriTeam execution engine, and discuss the CoVeriTeam program for our validating verifier in [Listing 1](#).

## 3 Design and Implementation of CoVeriTeam

We now explain details about the design and implementation of CoVeriTeam. First we discuss conceptual notions of actors, artifacts, and compositions; then we discuss execution concerns that a cooperative verification tool needs to handle. Then we delve deeper into implementation details where we discuss how an actor is created and executed. Last, we briefly explain the API that CoVeriTeam exposes and extensibility of this API.

### 3.1 Concepts

This section describes the language that we have designed for cooperative verification and on-demand composition. At first we describe the notion of artifacts and actors, and then the composition language to compose components to new actors.

**Artifacts and Actors.** Verification artifacts provide the means of information (and knowledge) exchange between the verification actors (tools). [Figure 1](#) shows a hierarchy of artifacts, restricted to those that we have used in the case studies for evaluating our work. On a high level we divide verification artifacts in

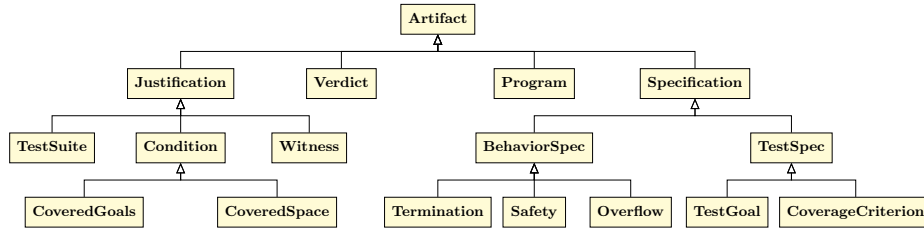


Fig. 1: Hierarchy of Artifacts (arrows indicate an is-a relation)

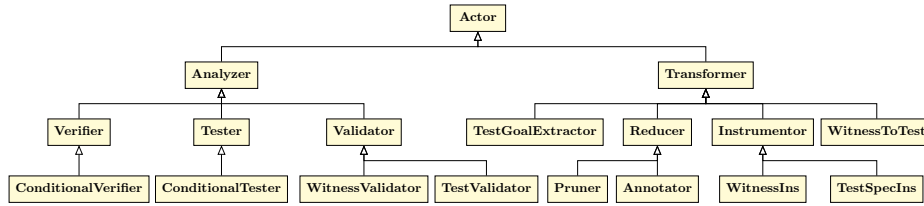


Fig. 2: Hierarchy of Actors (arrows indicate an is-a relation)

the following kinds: *Programs*, *Specifications*, *Verdicts*, and *Justifications*. *Programs* are behavior models (can be further classified into programs, control-flow graphs, timed automata, etc.). *Specifications* include behavioral specifications (for formal verification) and test specifications (coverage criteria for test-case generation). *Verdicts* are produced by actors signifying the class of result obtained (TRUE, FALSE, UNKNOWN, TIMEOUT, ERROR). *Justifications* for the verdict are produced by an actor; they include test suites to justify an obtained coverage, or verification witnesses to justify a verification result.

Verification actors act on the artifacts and as a result either produce new artifacts or transform a given artifact for consumption by some other actor. Figure 2 shows a hierarchy of actors, restricted to those that we have used in the case studies for evaluating our work. We divide verification actors in the following types: *Analyzers* and *Transformers*. *Analyzers* create new knowledge, e.g., verifiers, validators, and test generators. *Transformers* instrument, refine, or abstract artifacts.

**Composition.** Actors can be composed to create new actors. Our language supports the following compositions: *sequence*, *parallel*, *if-then-else*, and *repeat*.

CoVeriTEAM infers types and type-checks the compositions, and then either constructs a new actor or throws a type error. In the following, we use the notation  $I_a$  for the input parameter set of an actor  $a$  and  $O_a$  for the output parameter set of  $a$ . A parameter is a pair of name and *artifact type*. A *name clash* between two sets  $A$  and  $B$  exists if there is a name in  $A$  that is mapped to a different artifact type in  $B$ , more formally:  $\exists(a, t_1) \in A, (a, t_2) \in B : t_1 \neq t_2$ . The *actor type* is a mapping from input parameter set to output parameter set ( $I_a \rightarrow O_a$ ).

*Sequential.* Given two actors  $a1$  and  $a2$ , the sequential composition **SEQUENCE** ( $a1, a2$ ) (Fig. 3a) constructs an actor that executes  $a1$  and  $a2$  in sequence, i.e., one after another. The composition is well-typed if there is no name clash between  $I_{a1}$  and  $(I_{a2} \setminus O_{a1})$ . This means that we allow same artifact to be passed to the second actor in sequence, but disallow the confusing scenario

6 Dirk Beyer and Sudeep Kanav

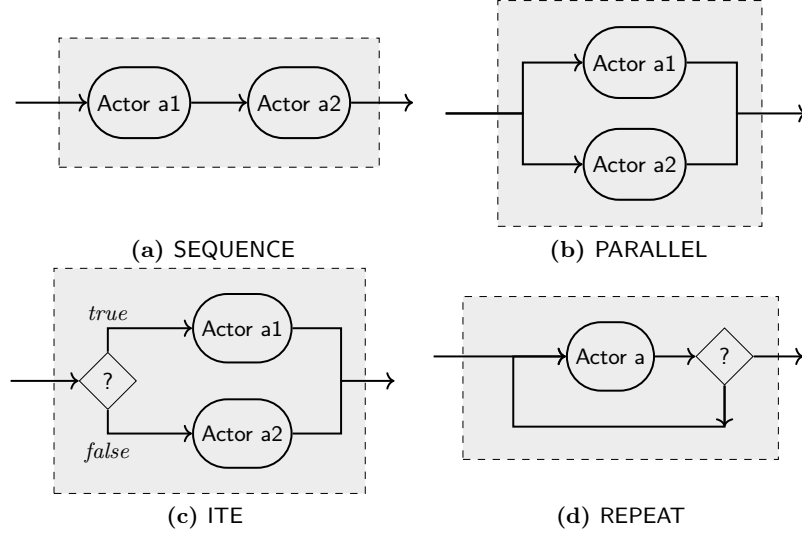


Fig. 3: Compositions in CoVeriTeam

where both actors expect an artifact with the same name but different type. The inferred type of the composition is  $I_{a1} \cup (I_{a2} \setminus O_{a1}) \rightarrow O_{a2}$ .

*Parallel.* Given two actors  $a1$  and  $a2$ , the parallel composition **PARALLEL** ( $a1, a2$ ) (Fig. 3b) constructs an actor that executes the actors  $a1$  and  $a2$  in parallel. The composition is well-typed if (a) there is no name clash between  $I_{a1}$  and  $I_{a2}$  and (b) the names of  $O_{a1}$  and  $O_{a2}$  are disjoint. The inferred type of the composition is  $I_{a1} \cup I_{a2} \rightarrow O_{a1} \cup O_{a2}$ .

*ITE.* Given a predicate  $cond$  and two actors  $a1$  and  $a2$ , the *if-then-else* composition **ITE** ( $cond, a1, a2$ ) (Fig. 3c) constructs an actor that executes the actor  $a1$  if predicate  $cond$  evaluates to *true*, and the actor  $a2$  otherwise. The composition is well-typed if (a) there is no name clash between  $cond$ ,  $I_{a1}$ , and  $I_{a2}$ , and (b) the output parameters are the same ( $O_{a1} = O_{a2}$ ). The inferred type of the composition is  $I_{a1} \cup I_{a2} \cup vars(cond) \rightarrow O_{a1}$ , where  $vars$  maps the variables used in a predicate to their artifact types. This allows us to define the condition  $cond$  using artifacts other than the inputs of  $I_{a1}$  and  $I_{a2}$ .

There are situations where  $a2$  is not required and its explicit specification only increases complexity. So, we have relaxed the type checker and made  $a2$  optional.

*Repeat.* Given a set  $fp$  and an actor  $a$ , the *repeat* composition **REPEAT** ( $fp, a$ ) (Fig. 3d) constructs an actor that repeatedly executes actor  $a$  until a fixed-point of set  $fp$  is reached, that is,  $fp$  did not change in the last execution of  $a$ . The *repeat* composition feeds back the output of  $a$  from iteration  $n$  to  $a$  for iteration  $n + 1$ . Let us partition  $I_a \cup O_a$  into three sets:  $I_a \setminus O_a$ ,  $O_a \setminus I_a$ , and  $I_a \cap O_a$ . The parameters in  $I_a \setminus O_a$  do not change their value and the parameters in  $O_a \setminus I_a$  are accumulated if accumulatable, otherwise their value after the execution of the composition is the value from the last iteration. The composition is well-typed if  $fp \subseteq dom(I_a \cap O_a)$ , where  $dom$  returns the names of a parameter set. The inferred type of the composition is  $I_a \rightarrow O_a$ .



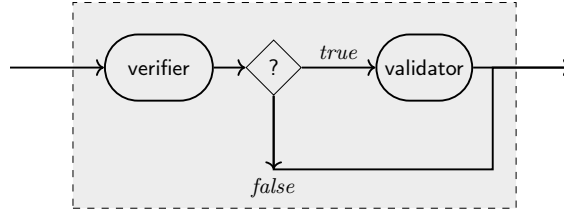


Fig. 4: CoVeriTEAM implementation of the validating verifier from Example 1

Figure 4 shows the pictorial representation of our running example using these compositions. First a verifier is executed, then the validator is executed if the verifier returned TRUE or FALSE, otherwise (in case of UNKNOWN, TIMEOUT, ERROR) the validator is not executed and the output of the verifier is forwarded.

### 3.2 Execution Concerns

A tool for cooperative verification orchestrates the execution of verification tools. This means it needs to assemble the command for the tool, as well as handle the output produced by the tool. A verification tool might consume a lot of resources and a user might want to limit this. It might crash during execution, might interfere with other processes. CoVeriTEAM needs to handle all these concerns.

Instead of developing our own infrastructure to handle these concerns, we reuse some of the features provided by BENCHEXEC [20]: we use tool-info modules to assemble command lines and parse log output, RUNEXEC (a component of BENCHEXEC) to execute tools in a container and limit resource consumption.

*Tool-Info modules* are integration modules of the benchmarking framework BENCHEXEC [20]. A typical tool-info module is a few lines of code used for assembling a command line and parsing the log output produced by the tool. It takes only a few hours to create one.<sup>1</sup> CoVeriTEAM uses tool-info modules to pass artifacts to atomic actors (assemble command-line) and extract artifacts from the output produced by the atomic actor. Using tool-info modules gave us integration of more than 80 tools without effort, because such integration modules exist for most well-known verifiers, validators, and testers (as many researchers use BENCHEXEC and provide such integration modules for their tools).

CoVeriTEAM uses RUNEXEC to isolate tool execution to prevent interference with the execution environment and enforce resource limits. We also report back to the user the resources consumed by the tool execution as measured by RUNEXEC.

### 3.3 CoVeriTEAM

Figure 5 provides an abstract view of the system. CoVeriTEAM takes as input a program written in the CoVeriTEAM language and artifacts. At first, the code generator converts this input program to Python code. This transformed

<sup>1</sup> We claim this based on our experience with tool developers creating their tool-info modules, which is a prerequisite for participating in SV-COMP or Test-Comp.



8 Dirk Beyer and Sudeep Kanav

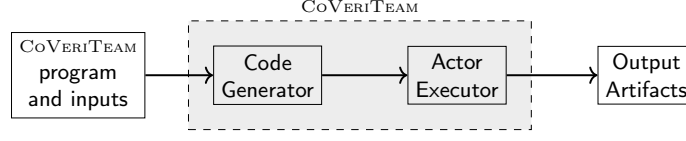


Fig. 5: Abstract view of the CoVeriTEAM tool

```

1 verifier = ActorFactory.create(ProgramVerifier,
    "actors/uautomizer.yml");
2 validator = ActorFactory.create(ProgramValidator,
    "actors/cpa-validate-violation-witnesses.yml");
3
4 // Use validator if verdict is true or false
5 condition = ELEMENTOF(verdict, {TRUE, FALSE});
6 second_component = ITE(condition, validator);
7 // Verifier and second component to be executed in sequence
8 validating_verifier = SEQUENCE(verifier, second_component);
9
10 // Prepare example inputs
11 prog = ArtifactFactory.create(CProgram, prog_path);
12 spec = ArtifactFactory.create(BehaviorSpecification, spec_path);
13 inputs = {'program':prog, 'spec':spec};
14 // Execute the new component on the inputs
15 res = execute(validating_verifier, inputs);
16 print(res);
  
```

Listing 1: CoVeriTEAM implementation of the validating verifier from Example 1

code uses the internal API of CoVeriTEAM. Then this Python code is executed, which means the actor executor is called on the specified actor. This in turn produces output artifacts on successful execution of the actor.

There are four key parts of executing a CoVeriTEAM program: creation of *atomic actors*, composition of *actors* (atomic or composite), creation of *artifacts*, and execution of the actors. We now give a brief explanation of these parts with the help of our running example. Listing 1 shows a CoVeriTEAM implementation of the running example (Example 1).

**Creation of an Atomic Actor.** Atomic actors in CoVeriTEAM provide an interface for external tools. CoVeriTEAM uses the information provided in an actor-definition file to construct an atomic actor. Lines 1 and 2 in Listing 1 show the creation of atomic actors `verifier` and `validator` using the *ActorFactory* by providing the *ActorType* and the actor-definition file. Once constructed, this actor can be executed.

An *actor definition* is specified in a file in YAML format. It contains the information necessary for executing the actor: location from where to download the tool, the name of the tool-info module to assemble the command line and parse tool output, additional command-line parameters for the tool, resource limitations to enforce, etc. Listing 2 shows the actor definition file for UAutomizer [32]: the actor name is `uautomizer`, the identifier for the BENCHEXEC tool-info module is

```

1 actor_name: uautomizer
2 toolinfo_module: "ultimateautomizer.py"
3 archive:
4   doi: "10.5281/zenodo.3813788"
5   spdx_license_identifier: "LGPL-3.0-or-later"
6 options: ['--full-output', '--architecture', '32bit']
7 resource_limits:
8   memlimit: "15 GB"
9   timelimit: "15 min"
10  cpuCores: "8"
11 format_version: '1.1'

```

Listing 2: Definition of atomic actor in YAML format

**ultimateautomizer**, the DOI of the tool archive (or the URL for obtaining the tool archive), the SPDX license identifier, the options passed by CoVeriTeam to UAutomizer, and resource limits for the execution of the actor. Once an atomic actor has been constructed using an actor definition, CoVeriTeam has all the information necessary to execute the underlying tool with the provided artifacts.

**Composition of an Actor.** The second key part is the composition of an actor. Lines 6 and 8 in Listing 1 create composite actors using ITE and SEQUENCE, respectively. It is these compositions that create the validating verifier of our running example. Verification actors in CoVeriTeam can exchange information (artifacts) with other actors and cooperate through compositions.

**Creation of an Artifact.** The notion of artifact in CoVeriTeam is a *file* wrapped in an *artifact type*. The underlying files are the basis of an artifact—exchangeable information. Lines 11 and 12 in Listing 1 create artifacts using the *ArtifactFactory* by providing the *ArtifactType* and the artifact file. These artifacts would then be provided to the executor that then executes the actors on them.

**Code Generation.** The code generator of CoVeriTeam translates the input program to Python code that uses the internal API of CoVeriTeam. It is a canonical transformation in which the statements for creation of actors and artifacts are converted to Python statements instantiating corresponding classes from the API. Similarly, statements for composition and execution of actors are also transformed.

**Execution.** Analogously to the construction of actors, the execution of an actor in CoVeriTeam is also divided in two: atomic and composition. Line 15 in Listing 1 executes the actor **validating\_verifier** on the given input artifacts.

Figure 6 shows the actor executor for both atomic and composite actors. It executes an actor on the provided artifacts. At first it type checks the inputs, i.e., check if the input types provided to actor comply with the expected input types of the actor. It then calls the executor for atomic or composite actor depending on the actor type. Thereafter, it type checks the outputs, and at last returns the artifacts.

Execution of an atomic actor means the execution of the underlying tool on the provided artifacts. At first, the executor downloads the tool if necessary. CoVeriTeam downloads and unzips the archive that contains the tool on the first execution of an atomic actor. It keeps the tool available in cache for later

10 Dirk Beyer and Sudeep Kanav

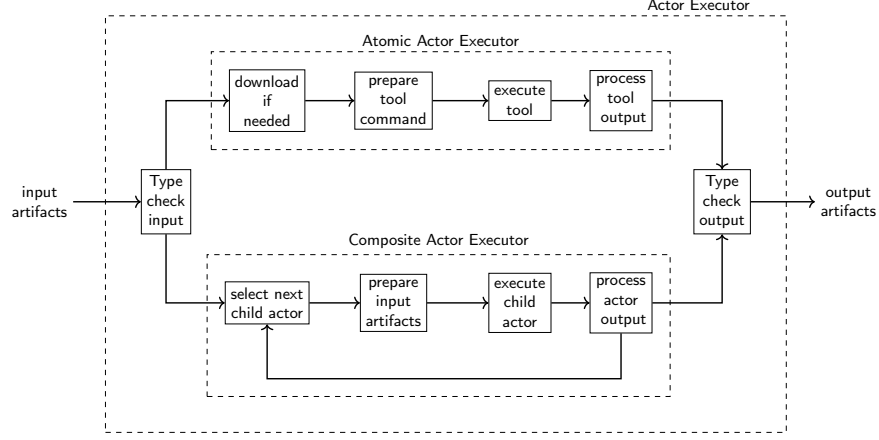


Fig. 6: Abstract view of an actor execution in CoVeriTEAM

executions. After this step, the command line for the tool is prepared using the tool-info module. It then executes the tool in a container, and then processes the tool output, i.e., extracts the artifacts from the tool output and saves them.

Execution of a composition means executing the composed actors—making information produced by one available to other during the execution—as per the rules of composition. The composite-actor executor at first selects the next child actor to execute. It then computes the inputs for this selected actor. Then it executes this actor, which can be atomic or another composite actor, on these inputs. It then processes the outputs produced by the execution of the selected child actor. This processing could be temporarily saving, filtering, or modifying the produced artifacts. If needed, it then proceeds to execute the next child actor, otherwise exits the composition execution.

**Output.** CoVeriTEAM collects all the artifacts produced during the execution of an actor, and saves them. The output can be divided into three parts: execution trace, artifacts, and log files. An execution trace is an XML file containing information about the artifacts consumed and produced by each actor, and also the resources consumed by atomic actors (as measured by BENCHEXEC) during the execution. CoVeriTEAM also saves the artifacts produced during the execution of an actor. Additionally, for each atomic actor execution, it also saves a log file containing the command which was actually executed and the messages printed on *stdout*.

### 3.4 API

In addition to the above described features, CoVeriTEAM exposes an API that is extensible. We expose actors, artifacts, utility actors, and compositions through Python packages. In this section, we briefly discuss this API.

**Library of Actors and Compositions.** CoVeriTEAM provides a library of some actors and a few compositions that can be instantiated with suitable

actors. We considered actors based on the tools participating in the competitions on software verification and testing [5, 6] (available in the replication archives), because those are known to be mature and stable.

The library of compositions contains a validating verifier, an execution-based validator [11], a reducer-based construction of a conditional model checker [15], CONDTEST [18], and METAVAL [21]. These are present in the `examples/` directory of the CoVeriTeam repository. We discuss some of these constructions in Sect. 4.1.

**New Actors, Artifacts, and Tools.** New actors, artifacts, and tools can be integrated easily in CoVeriTeam. The integration of a new atomic actor requires only creating a YAML actor definition and, if not already available, implementing a tool-info module. The integration of a new actor type in the language requires (1) creating a class for the actor specifying its input and output artifact types, (2) preparing the parameters to be passed to tool-info module, that in turn would create a command line for the tool execution, using the options from the YAML actor definition, and (3) creating output artifacts from the output files produced by the execution of an atomic actor of that type.

Integration of a new artifact requires creating a new class for the artifact. A basic artifact requires a path containing the artifact. Some artifacts support special features, for example, a test suite is a mergeable artifact (i.e., two test suites for a given input program can be merged into one test suite).

Integrating a new tool in the framework requires: (1) creating the tool-info module for it, (2) creating an actor definition for the tool, (3) providing a self-contained archive that can be executed on a Ubuntu machine.

At present, CoVeriTeam supports all verifiers and validators that are listed on the 2021 competition web sites of SV-COMP<sup>2</sup> and Test-Comp<sup>3</sup>. One needs only a few hours to create a new tool-info module and an actor-definition file. Within a couple of hours we were able to create the actor definitions for about 40 tools participating in SV-COMP and Test-Comp.

## 4 Evaluation

We now present our evaluation of CoVeriTeam. It consists of a few case studies, and insights from the experiments to measure performance overhead.

### 4.1 Case Studies

We evaluated CoVeriTeam on four more case studies, as indicated in the fourth column of Table 1. We now explain two of these case studies using figures for compositions. The programs and explanations for all of the case studies are also available in our project repository (linked from the last column of Table 1).

**Conditional Testing à la CONDTEST.** Conditional testing [18] allows cooperation between different test generators (testers) by sharing the details of the

<sup>2</sup> <https://sv-comp.sosy-lab.org/2021/systems.php>

<sup>3</sup> <https://test-comp.sosy-lab.org/2021/systems.php>

12 Dirk Beyer and Sudeep Kanav

Table 1: Examples of cooperative techniques in the literature

Technique	Year	Reference	Case Study	More Info
Counterexample Checking [38]	2012	<a href="#">Sect. 5</a>		
Conditional Model Checking [13]	2012	<a href="#">Sect. 5</a>		
Precision Reuse [19]	2013	<a href="#">Sect. 5</a>		
Witness Validation [8, 10]	2015, 2016	<a href="#">Figure 4</a>	✓	<a href="#">Sect. 3.3</a>
Execution-Based Validation [11]	2018	<a href="#">Sect. 5</a>	✓	<a href="#">More info</a>
Reducer [15]	2018	<a href="#">Sect. 5</a>	✓	<a href="#">More info</a>
CoVeriTest [14]	2019	<a href="#">Sect. 5</a>		
CONDTEST [18]	2019	<a href="#">Figures 7 and 8</a>	✓	<a href="#">More info</a>
METAVAL [21]	2020	<a href="#">Figure 9</a>	✓	<a href="#">More info</a>

already covered test goals. A conditional tester outputs a condition, in addition to the generated test suite, representing the work already done. Then this condition is passed as an input to another conditional tester, in addition to the program and test specification. This tester can then focus on only the uncovered goals.

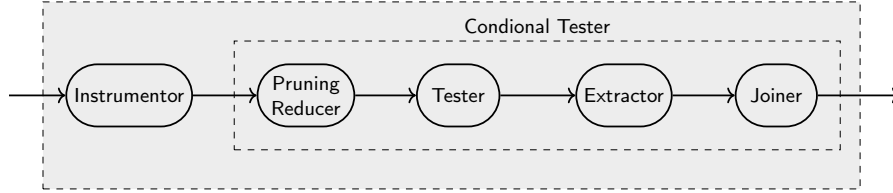


Fig. 7: Design of a conditional tester in CoVeriTEAM

Conditional testers can be constructed from off-the-shelf testers [18] with the help of three tools: a reducer, an extractor, and a joiner. A reducer used in conditional testing ( $\text{Program} \times \text{Specification} \times \text{Condition} \rightarrow \text{Program}$ ) produces a residual program with the same behavior as the input program with respect to the remaining test goals. A set of test goals represents the condition. An extractor ( $\text{Program} \times \text{Specification} \times \text{TestSuite} \rightarrow \text{Condition}$ ) extracts the condition—a set of test goals—covered by the provided test suite.

Figure 7 shows the composition of a conditional tester. First, the reducer produces the reduced program. The composition here uses a *pruning* reducer, which prunes the program according to the covered goals. Second, the tester generates the test cases. Third, the extractor extracts the goals covered in these test cases. Forth, the joiner merges the previously and newly covered goals. The reducer that we used expects the input program to be in a format containing certain labels for the purpose of tracking test goals. So, we put an instrumentor that *instruments* the test specification into the program, by adding these labels.

The conditional-testing concept can also be used iteratively to generate a test suite using a tester based on a verifier [18]. Such a composition uses a verifier as a backend and transforms a counterexample generated by the verifier to a test case.

Figure 8 shows the construction of a cyclic conditional tester. In this case, the *tester* itself is a composition of a verifier and a tool, *Witness2Test*, which generates test cases based on a witness produced by a verifier. This tester, in composition

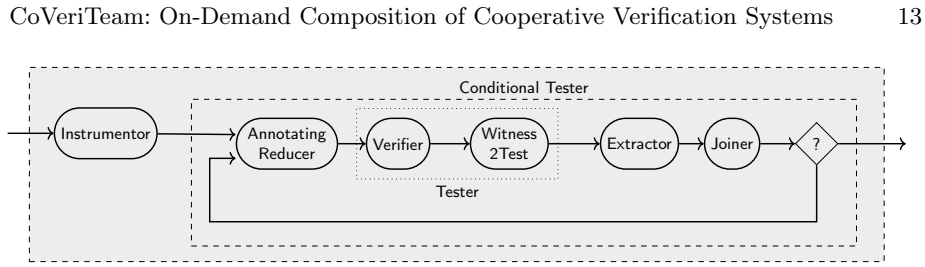


Fig. 8: Design of a cyclic conditional tester in CoVeriTeam

with a reducer, extractor, and a joiner is our conditional tester. This construction uses an *annotating* reducer, which (i) annotates the program with *error* labels for the verifier to find the path to and (ii) filters out the already covered goals, i.e., the condition, from the list of goals to be annotated. We put the conditional tester in the REPEAT composition to execute iteratively. The composition tracks the set ‘*covered\_goals*’ to detect the fixed point to decide termination of the iteration. This composition will keep on accumulating the test suite generated in each iteration and finally output the union of all the generated test suites (see Sect. 3.1). As above, an instrumentor is placed before the conditional tester.

**Verification-Based Validation à la METAVAL.** METAVAL [21] uses off-the-shelf verifiers to perform validation tasks. A validator ( $\text{Program} \times \text{Specification} \times \text{Verdict} \times \text{Witness} \rightarrow \text{Verdict} \times \text{Witness}$ ) validates the result produced by a verifier. METAVAL employs a three-stage process for validation. In the first stage, METAVAL instruments the input program with the input witness. The instrumented program—a product of the witness and the original program—is equivalent to the original program modulo the provided witness. This means that the instrumented program can be given to an off-the-shelf verifier for verification; and this verification functions as validation. In the second stage, METAVAL selects the verifier to use based on the specification. It chooses CPACHECKER for reachability, UAUTOMIZER for integer overflow and termination, and SYMBIOTIC for memory safety.<sup>4</sup> In the third stage, the instrumented program is fed to a verifier along with the specification for verification. If the verification produces the expected result, then the result is confirmed and the witness valid, otherwise not.

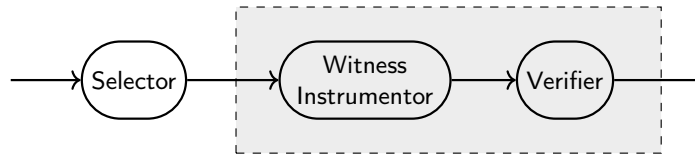


Fig. 9: Design of METAVAL in CoVeriTeam

Figure 9 shows the construction of METAVAL. First, the selector is executed that selects the backend verifier to execute. After this step, the program is

<sup>4</sup> These were the best performing tools for a property according to SV-COMP results.

14 Dirk Beyer and Sudeep Kanav

instrumented with the witness, and then the instrumented program is given to the selected verifier for checking the specification.

## 4.2 Performance

CoVeriTeam is a lightweight tool. Its container mode causes an overhead of around 0.8s for each actor execution in the composition, and the tool needs about 44 MB memory. This means that if we run a tool 10 times in a sequence in a shell script unprotected and compare this to using the sequence composition in CoVeriTeam in protected container mode on the same input, the execution using CoVeriTeam will take 8s longer and requires 44 MB more memory. In our experience, this overhead is not an issue for verification as, in general, the time taken for verification dominates the total execution time. For short-running, high-performance needs, the container mode can be switched off. We have conducted extensive experiments for performance evaluation of CoVeriTeam and point the reader to the [supplementary webpage](#) for this article for more details.

## 5 Related Work

We divide our literature overview into two parts: approaches for tool combinations, and cooperative verification approaches.

**Approaches for Tool Combinations.** *Evidential Tool Bus* (ETB) [29, 30, 39] is a distributed framework for integration of tools based on a variant of Datalog [1, 24]. It stores the established claims along with the corresponding files and their versions. This allows the reuse of partial results in regression verification. ETB orchestrates tool interaction through scripts, queries, and claims.

Our work seems close to ETB on a quick glance, but on a closer look there are profound differences. Conceptually, ETB is a query engine that uses claims, facts, and rules to define and execute a workflow. Whereas, CoVeriTeam has been designed to create and execute actors based on tools and their compositions. We give some semantic meaning, arguably simplistic, to the tools using (i) wrapper types of artifacts for the files produced and consumed by a tool and (ii) the notion of *verification actors* that allows us to see a tool as a *function*. This allows us to type-check tool compositions and allow only well-defined compositions. On the implementation side, we support more tools. This task was simplified by our design choice to use the integration mechanisms provided by BENCHEXEC (as used in SV-COMP and Test-Comp). Most well known automated verification tools already have been integrated in CoVeriTeam.

*Electronic Tools Integration* platform (ETI) [40] was envisioned as a “one stop shop” for the experimentation and evaluation of tools from the formal-methods community. It was intended to serve as a tool presentation, tool evaluation, and benchmarking site. The idea was to allow users to access tools through the internet without the need to install them. An ETI user is expected to provide an LTL based specification, based on which an execution scheme is synthesized.

The key focus of ETI and its incarnations has been remote tool execution, and their integration over internet. The tools are viewed agnostic to their function. We, in contrast, (i) have tackled local execution concerns and (ii) see a tool in its function as an actor that consumes and produces certain kinds of artifacts. The semantic meaning of a tool is given by this role.

**Cooperative Verification Approaches.** Our work targets developing a framework to express and execute cooperative verification approaches. In this section we describe some of these approaches from literature. We have implemented some of these combinations in CoVeriTeam, some of which are described in Sect. 4.

A reduction of the input program using the counterexample produced by a verifier was discussed [38], where the key idea is to use the counterexample to provide the variable assignments to the program.

Conditional model checking (CMC) [13] outputs a condition—a summary of the knowledge gained—if the model checker fails to produce a verdict. The condition allows another model checker to save the effort of looking into already explored state space. Reducers [15] can turn any off-the-shelf model checker into a conditional model checker. Reducers take a source program and a condition and produce a *residual program* whose paths cover the unverified state space (negation of the condition). Conditional testing [18] applies the principle of conditional model checking to testing. A conditional tester outputs, in addition to the generated test cases, the goals for which test cases have been generated.

The idea of reusing the knowledge about already done work to reduce the workload of another tool was also applied to combine program analysis and testing [25, 31, 35]. One of these approaches [31] is based on conditional model checking [13]. In this case, the condition is used to construct a residual program, which is then fed to a test-case generator. Another approach [25] instruments the program with assumptions and assertions describing the already completed verification work. Then a testing tool is used to test the assumptions. Program partitioning [35] first performs the testing and then removes the satisfactorily tested paths and verifies the rest. CoVeriTest [14], cooperative verifier-based testing, is a tester based on cooperation between different verification-based test-generation techniques. CoVeriTest uses conditional model checkers [13] as verifier backends.

Precision reuse [19] is based on the use of abstraction precisions. The precision of an abstract domain is a good candidate for cooperation because it is small in size, and represents important information, i.e., the level of abstraction at which the analysis works. A model checker in addition to producing a verdict also produces a file containing information specifying precision, e.g., predicates.

Model checkers can also produce a witness, in addition to the verdict, as a justification of the verdict. These witnesses could be counterexamples for violations of a safety property, invariants as a proof of a safety property, a lasso for non-termination, a ranking function for termination, etc. These witnesses can be used later to help validate the result produced by a verifier [8, 9, 10].

Execution-based result validation [11] uses violation witnesses to generate test cases. A violation witness of a safety specification is refined to a test case. The test case is then used to validate the result of the verification.



16 Dirk Beyer and Sudeep Kanav

## 6 Conclusion

Due to the free availability of many excellent verifiers, the time is ripe to view verification tools as components. It is necessary to have standardized interfaces, in order to define the inputs and outputs of verification components. We have identified a set of verification artifacts and verification actors, and a programming language for on-demand construction of new, combined verification systems.

So far, the architectural hierarchy ends mostly at the verifiers: verifiers are based on SMT solvers, which are based on SAT solvers, which are based on data-structure libraries. CoVeriTeam wants to change this and use verification artifacts as first-class objects in specifying new verifiers. We show on a few selected examples how easy it is to construct some verification systems that were so far hard-coded using glue code and wrapper scripts. We hope that many researchers and practitioners in the verification community find it interesting and stimulating to experiment on a high level with verification technology.

*Future Work.* The approach of CoVeriTeam opens up a whole new area of possibilities that yet needs to be explored. We have identified three key areas for the further work: (i) remote execution of tools, (ii) policy specification and enforcement, and (iii) more compositions and combinations. CoVeriTeam provides an interface for a verification tool based on its behavior. A web service wrapped around CoVeriTeam can be used to delegate execution of an actor, hence verification work, to the host of the service. The client for such a service can be transparently integrated in CoVeriTeam. In fact, we already provide client integration for a restricted and experimental version of such a service. Also, a user executing a combination of tools might want to have some restrictions on which tools should be allowed to execute. For example, a user might want to execute only those tools that comply with a certain license, or only those tools that are downloaded from a trusted source. A cooperative verification tool should support the specification and enforcement of such user *policies*. Further, we plan to support more compositions for cooperative verification in CoVeriTeam as we come across them. Recently, we were working on a *parallel-portfolio* composition [17].

## Declarations

**Data Availability Statement.** CoVeriTeam is publicly available under the Apache 2 license.<sup>5</sup> The data from our performance evaluation is available at the supplementary webpage of the paper.<sup>6</sup> A replication package including all evaluated external tools is available at Zenodo [16].

**Funding Statement.** This work was funded in part by the Deutsche Forschungsgesellschaft (DFG) — 418257054 (Coop).

**Acknowledgement.** We thank Thomas Lemberger and Philipp Wendler for their valuable feedback on this article, and the SV-COMP community for their valuable feedback on experimenting with CoVeriTeam.

<sup>5</sup> <https://gitlab.com/sosy-lab/software/coveriteam/>

<sup>6</sup> <https://www.sosy-lab.org/research/coveriteam/>

## References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: Proc. IFM. pp. 1–20. LNCS 2999, Springer (2004). [https://doi.org/10.1007/978-3-540-24756-2\\_1](https://doi.org/10.1007/978-3-540-24756-2_1)
3. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM **54**(7), 68–76 (2011). <https://doi.org/10.1145/1965724.1965743>
4. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.5. Tech. rep., University of Iowa (2015), available at [www.smt-lib.org](http://www.smt-lib.org)
5. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Proc. TACAS (2). pp. 347–367. LNCS 12079, Springer (2020). [https://doi.org/10.1007/978-3-030-45237-7\\_21](https://doi.org/10.1007/978-3-030-45237-7_21)
6. Beyer, D.: Second competition on software testing: Test-Comp 2020. In: Proc. FASE. pp. 505–519. LNCS 12076, Springer (2020). [https://doi.org/10.1007/978-3-030-45234-6\\_25](https://doi.org/10.1007/978-3-030-45234-6_25)
7. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS. LNCS 13244, Springer (2022)
8. 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>
9. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. (2022)
10. 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>
11. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP. pp. 3–23. LNCS 10889, Springer (2018). [https://doi.org/10.1007/978-3-319-92994-1\\_1](https://doi.org/10.1007/978-3-319-92994-1_1)
12. Beyer, D., Haltermann, J., Lemberger, T., Wehrheim, H.: Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR. In: Proc. ICSE. ACM (2022)
13. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE. ACM (2012). <https://doi.org/10.1145/2393596.2393664>
14. Beyer, D., Jakobs, M.C.: CoVeriTest: Cooperative verifier-based testing. In: Proc. FASE. pp. 389–408. LNCS 11424, Springer (2019). [https://doi.org/10.1007/978-3-030-16722-6\\_23](https://doi.org/10.1007/978-3-030-16722-6_23)
15. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). <https://doi.org/10.1145/3180155.3180259>
16. Beyer, D., Kanav, S.: Reproduction package for article ‘CoVeriTeam: On-demand composition of cooperative verification systems’. Zenodo (2021). <https://doi.org/10.5281/zenodo.5644953>
17. Beyer, D., Kanav, S., Richter, C.: Construction of Verifier Combinations Based on Off-the-Shelf Verifiers. In: Proc. FASE. Springer (2022)
18. Beyer, D., Lemberger, T.: Conditional testing: Off-the-shelf combination of test-case generators. In: Proc. ATVA. pp. 189–208. LNCS 11781, Springer (2019). [https://doi.org/10.1007/978-3-030-31784-3\\_11](https://doi.org/10.1007/978-3-030-31784-3_11)

18 Dirk Beyer and Sudeep Kanav

19. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proc. FSE. pp. 389–399. ACM (2013). <https://doi.org/10.1145/2491411.2491429>
20. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. *Int. J. Softw. Tools Technol. Transfer* **21**(1), 1–29 (2019). <https://doi.org/10.1007/s10009-017-0469-y>
21. Beyer, D., Spiessl, M.: METAVAL: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). [https://doi.org/10.1007/978-3-030-53291-8\\_10](https://doi.org/10.1007/978-3-030-53291-8_10)
22. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1). pp. 143–167. LNCS 12476, Springer (2020). [https://doi.org/10.1007/978-3-030-61362-4\\_8](https://doi.org/10.1007/978-3-030-61362-4_8)
23. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O’Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. pp. 3–11. LNCS 9058, Springer (2015). [https://doi.org/10.1007/978-3-319-17524-9\\_1](https://doi.org/10.1007/978-3-319-17524-9_1)
24. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. Knowledge and Data Eng.* **1**(1), 146–166 (1989). <https://doi.org/10.1109/69.43410>
25. Christakis, M., Müller, P., Wüstholtz, V.: Collaborative verification and testing with explicit assumptions. In: Proc. FM. pp. 132–146. LNCS 7436, Springer (2012). [https://doi.org/10.1007/978-3-642-32759-9\\_13](https://doi.org/10.1007/978-3-642-32759-9_13)
26. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proc. CAV. pp. 154–169. LNCS 1855, Springer (2000). [https://doi.org/10.1007/10722167\\_15](https://doi.org/10.1007/10722167_15)
27. Cook, B.: Formal reasoning about the security of Amazon web services. In: Proc. CAV (2). pp. 38–47. LNCS 10981, Springer (2018). [https://doi.org/10.1007/978-3-319-96145-3\\_3](https://doi.org/10.1007/978-3-319-96145-3_3)
28. Cousot, P., Cousot, R.: Systematic design of program-analysis frameworks. In: Proc. POPL. pp. 269–282. ACM (1979). <https://doi.org/10.1145/567752.567778>
29. Cruanes, S., Hamon, G., Owre, S., Shankar, N.: Tool integration with the Evidential Tool Bus. In: Proc. VMCAI. pp. 275–294. LNCS 7737, Springer (2013). [https://doi.org/10.1007/978-3-642-35873-9\\_18](https://doi.org/10.1007/978-3-642-35873-9_18)
30. Cruanes, S., Heymans, S., Mason, I., Owre, S., Shankar, N.: The semantics of Datalog for the Evidential Tool Bus. In: *Specification, Algebra, and Software*. pp. 256–275. Springer (2014). [https://doi.org/10.1007/978-3-642-54624-2\\_13](https://doi.org/10.1007/978-3-642-54624-2_13)
31. Czech, M., Jakobs, M., Wehrheim, H.: Just test what you cannot verify! In: Proc. FASE. pp. 100–114. LNCS 9033, Springer (2015). [https://doi.org/10.1007/978-3-662-46675-9\\_7](https://doi.org/10.1007/978-3-662-46675-9_7)
32. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). [https://doi.org/10.1007/978-3-642-39799-8\\_2](https://doi.org/10.1007/978-3-642-39799-8_2)
33. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. *Science* **275**(7), 51–54 (1997). <https://doi.org/10.1126/science.275.5296.51>
34. Inoue, K., Soh, T., Ueda, S., Sasaura, Y., Banbara, M., Tamura, N.: A competitive and cooperative approach to propositional satisfiability. *Discrete Applied Mathematics* **154**(16), 2291–2306 (2006). <https://doi.org/10.1016/j.dam.2006.04.015>
35. Jalote, P., Vangala, V., Singh, T., Jain, P.: Program partitioning: A framework for combining static and dynamic analysis. In: Proc. WODA. pp. 11–16. ACM (2006). <https://doi.org/10.1145/1138912.1138916>

36. Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: Proc. Ershov Memorial Conference. pp. 165–176. LNCS 5947, Springer (2009). [https://doi.org/10.1007/978-3-642-11486-1\\_14](https://doi.org/10.1007/978-3-642-11486-1_14)
37. Rice, J.R.: The algorithm selection problem. *Advances in Computers* **15**, 65–118 (1976). [https://doi.org/10.1016/S0065-2458\(08\)60520-3](https://doi.org/10.1016/S0065-2458(08)60520-3)
38. Rocha, H.O., Barreto, R.S., Cordeiro, L.C., Neto, A.D.: Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In: Proc. IFM. pp. 128–142. LNCS 7321, Springer (2012). [https://doi.org/10.1007/978-3-642-30729-4\\_10](https://doi.org/10.1007/978-3-642-30729-4_10)
39. Rushby, J.M.: An Evidential Tool Bus. In: Proc. ICFEM. pp. 36–36. LNCS 3785, Springer (2005). [https://doi.org/10.1007/11576280\\_3](https://doi.org/10.1007/11576280_3)
40. Steffen, B., Margaria, T., Braun, V.: The Electronic Tool Integration platform: Concepts and design. *STTT* **1**(1-2), 9–30 (1997). <https://doi.org/10.1007/s100090050003>

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Formal Methods in System Design  
<https://doi.org/10.1007/s10703-024-00449-y>



## Construction of verifier combinations from off-the-shelf components

Dirk Beyer<sup>1</sup> · Sudeep Kanav<sup>1</sup> · Tobias Kleinert<sup>1</sup> · Cedric Richter<sup>2</sup>

Received: 19 November 2022 / Accepted: 7 June 2023  
 © The Author(s) 2025

### Abstract

Software verifiers have different strengths and weaknesses, depending on the characteristics of the verification task. It is well-known that combinations of verifiers via portfolio- and selection-based approaches can help to combine their strengths. In this paper, we investigate (a) how to easily compose such combinations from *existing*, ‘off-the-shelf’ verifiers without changing them and (b) how much performance improvement each combination can yield, regarding the effectiveness (number of solved verification tasks) and efficiency (consumed resources). First, we contribute a method to systematically and conveniently construct verifier combinations from existing tools using COVERTEAM. We consider sequential portfolios, parallel portfolios, and algorithm selections. Second, we perform a large experiment to show that combinations can improve the verification results *without* additional computational resources. Our benchmark set is the category *ReachSafety* as used in the 11th Competition on Software Verification (SV-COMP 2022). This category contains 5400 verification tasks, with diverse characteristics. The key novelty of this work in comparison to the conference version of the article is to introduce a validation step into the verifier combinations. By validating the output of the verifier, we can mitigate the adverse effect of unsound tools on the performance of portfolios, especially parallel portfolios, as observed in our previous experiments. We confirm that combinations employing a validation process are significantly more robust against the inclusion of unsound verifiers. Finally, all combinations are constructed from off-the-shelf verifiers, that is, we use the verification tools as published. The results of our work suggest that users of combinations of verification tools can achieve a significant improvement at a negligible cost, and more robustness by using combinations with validators.

**Keywords** Software verification · Program analysis · Cooperative verification · Tool combinations · Portfolio · Algorithm selection · COVERTEAM · BENCHEXEC

---

✉ Dirk Beyer  
 dirk.beyer@sosy-lab.org  
<https://www.sosy-lab.org/people/beyer/>  
 Sudeep Kanav  
 sudeep.kanav@gmail.com

<sup>1</sup> LMU Munich, Munich, Germany

<sup>2</sup> Carl von Ossietzky University, Oldenburg, Germany

Published online: 11 March 2025

Springer

## 1 Introduction

Automatic software verification has been an active area of research since two decades [1], and various tools and techniques have been developed to solve the problem of verifying software [2–8]. The research has also been adopted in practice [9–12]. Each tool and technique has its own strengths in specific areas. In fact, an analysis of the results of category *ReachSafety* of SV-COMP 2022 [5] shows that even though high-performing tools such as CPAchecker [13, 14] and ESBMC [15] share tasks that both tools can solve, there is a significant number of tasks solved uniquely by one of the two tools (see Fig. 1). In such a scenario, it becomes obvious to combine these tools to benefit from the strengths of individual tools, leading to a ‘meta verifier’ that solves more verification tasks (e.g., up to 692 tasks more for a combination of CPAchecker and ESBMC). Most current combination approaches are hard-coded, that is, the choice of the tools to combine is fixed and the glue-code required to combine them is specifically programmed.

We contribute a method to construct combinations in a systematic way, independently from the set of tools to use. We considered the following three types of combinations: sequential and parallel portfolio [16], and algorithm selection [17]. In a sequential portfolio, the components are executed in sequence (one after another) until one of them succeeds (split time; full cores and memory; split risk). In a parallel portfolio, the components are executed in parallel until one of them solves the task (full time; split cores and memory; split risk). In algorithm selection, first, a selector selects a component that is most likely to solve the given task, and then only the chosen component is executed (full time; full cores and memory; full risk).

We use COVERITEAM [18–20] to construct and execute the combinations. COVERITEAM is a tool that is based on off-the-shelf atomic actors, which are executable units based on tool archives. It provides a simple language to construct tool combinations, and manages the download and execution of the existing tools on the provided input. COVERITEAM provides a library of atomic actors for many well-known and publicly available verification tools. A new verification tool can be easily integrated into COVERITEAM within a few minutes of effort.

This paper is an extended version of an article presented at FASE 2022 [21], with an attempt to mitigate its limitations. One of the limitations was that parallel portfolios are biased toward faster tools and would produce incorrect results if there is a fast but unsound tool included in the portfolio. We had mentioned two remedies for this issue: (i) add a validation step after the verification and (ii) carefully select the verifiers to include in a portfolio.

In this work, we apply the first remedy: we add a validation step to validate the results produced by a verifier. The verifiers that we use in our experiments also produce witnesses,

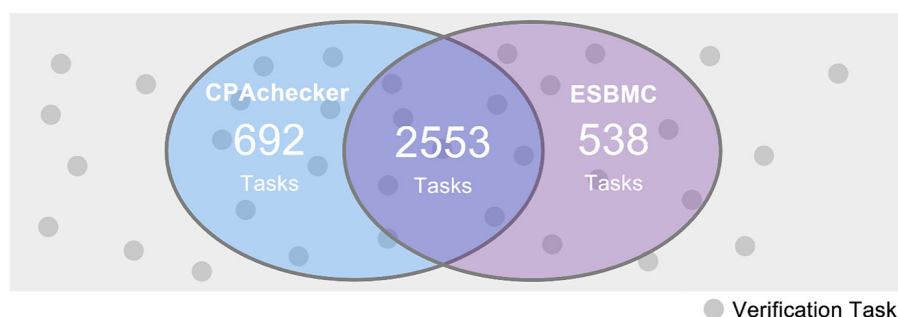


Fig. 1 Overlap of tasks solved by CPAchecker and ESBMC in SV-COMP 2022

in addition to the verdict, if the verification succeeds. Producing witnesses is a requirement to participate in SV-COMP, as the competition grants points only to those verifiers whose results can be validated. We use a subset of validators used in SV-COMP. For each VERIFIER we construct a VERIFIER<sup>Val</sup>: a sequence of a verifier and a validator. We then put this combination in sequential portfolios, parallel portfolios, and algorithm selections.

Our experiments are based on tools and benchmark verification tasks from the 11th Competition on Software Verification (SV-COMP 2022) [5].

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

1. We show how to conveniently construct combination approaches from off-the-shelf verification tools in a modular manner, without changing the tools.
2. We perform an extensive comparative evaluation of combination approaches based on sequential portfolios, parallel portfolios, and algorithm selections.
3. We provide a reproduction package containing tools and experiment data [22].

## 2 Overview of Combination Types for Off-the-Shelf Verifiers

In this study, we explore different types of combining verifiers to improve the overall verification effectiveness. We focus on the most common types of combinations that do not require any changes to the existing tools (off-the-shelf) or communication between the tools, which are: *sequential portfolio* [13, 23, 24], *parallel portfolio* [16, 25, 26], and *algorithm selection* [17, 27–30]. We now briefly describe these combination types and give an illustration in Fig. 2.

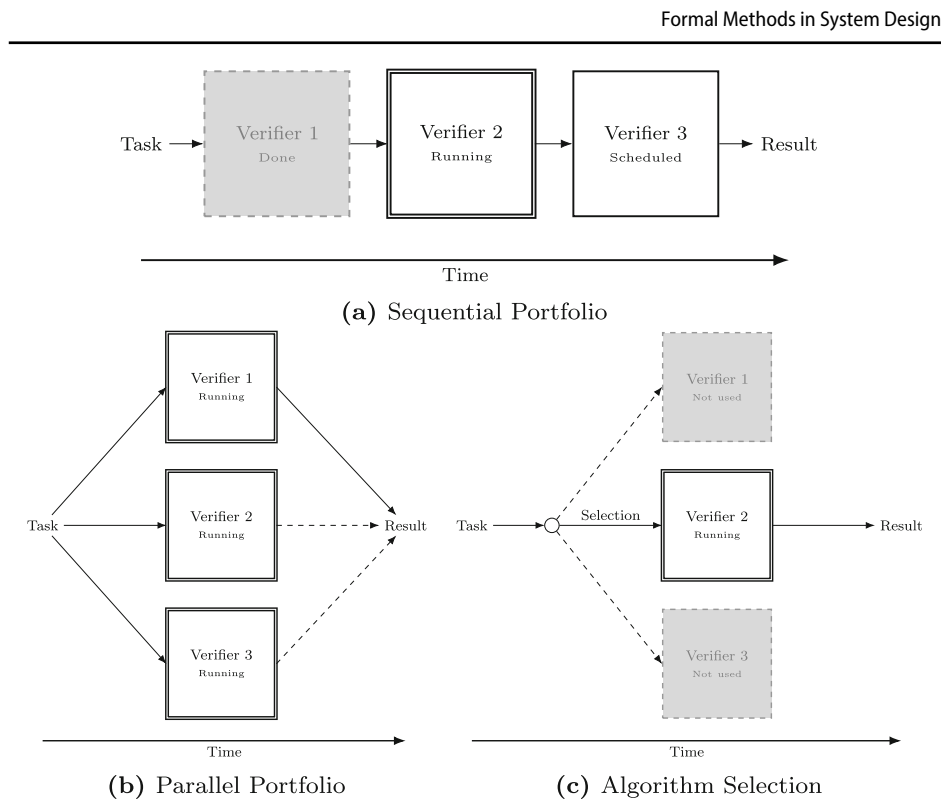
**Sequential Portfolio.** Portfolios combine several verifiers by executing them either sequentially or in parallel. A sequential portfolio (Fig. 2a) executes a set of verifiers in sequence, running them one after another. In this setting, each verifier is assigned a specific time limit and the verifier runs until it finds a solution or reaches the time limit. If the current verifier can solve the given verification task within the allocated time, the portfolio is stopped and the solution is emitted. Otherwise, if the current verifier runs into a timeout without solving the given task, it is terminated and the next one is started. CPA-Seq [13, 23] and Ultimate Automizer [24] are examples of a sequential portfolio.

**Parallel Portfolio.** In contrast to a sequential portfolio, a parallel portfolio (Fig. 2b) executes all verifiers in parallel, sharing all system resources like CPU cores and memory. As soon as one algorithm solves the given verification task, the portfolio is stopped and the solution is emitted. Since the physical computing resources are shared in a parallel portfolio, a tool may use up all its memory quota sooner than when running alone, and be terminated. PredatorHP [25, 26] is an example of a parallel portfolio.

**Algorithm Selection.** To reduce spending resources on unsuccessful verifiers, algorithm selection (Fig. 2c) is designed to select the verifier that is likely well suited to solve the given verification task. More precisely, algorithm selection first analyzes the given verification task for common characteristics, e.g., program features like the existence of a loop or an array. It then selects a verifier that is most likely to solve verification tasks with those characteristics. Then the selected verifier is executed. Algorithm selection was recently explored for selecting from a set of verification algorithms, e.g., in PeSCo [27, 28], or from a set of sequential portfolios of verification algorithms, e.g., in CPAchecker [29].

The above combination types have their own advantages and limitations when applied in real-world scenarios. While algorithm selection gives the full resources to one verifier, and thus, increases the chances that the verifier succeeds, it also takes the full risk of selecting





**Fig. 2** (a) A sequential portfolio runs each verifier for a certain maximal amount of time in a sequence. If a verifier stops with a result, the portfolio finishes. The available CPU time is split among the verifiers. (b) A parallel portfolio runs all verifiers simultaneously. If a verifier stops with a result, the portfolio finishes. The available CPU cores and memory are split among the verifiers. (c) An algorithm selection first selects a verifier and then executes it. The result produced by this verifier is taken. The selected verifier gets all available resources, and also the risk that the verifier does not deliver a result is not split

a sub-optimal verifier. If the selection algorithm is not powerful enough or the selection task is too difficult (i.e., the selection cannot be decided based on high-level features), the selector might fail to identify a verifier that is appropriate for the given task. Although portfolios omit this problem by assigning the verification task to several verifiers, each verifier gets fewer resources.

**Validation of the Verifier Results.** Verifiers can have bugs, hence, it is desirable to validate the result of the verification. One of the proposed options in the literature is that the verifier produces a justification of its verdict in the form of a verification witness in an exchangeable format [31–34]. A user can either inspect the witness manually [35] or use a tool to validate the result produced by the verifier.

Verification followed by result validation can also be achieved by a combination of a verifier and a validator. First, a verifier is executed to solve a verification task, then a validator is executed to validate the result of the verifier. We refer to such a combination as *validating verifier* (or VERIFIER<sup>Val</sup>). A validating verifier has the advantage that each emitted verification result is validated by an external validator.

Therefore, we can avoid emitting wrong results, which in turn can positively impact the effectiveness of verifier combinations.



Validators have been used in the competition on software verification since 2015 [36]. Intuitively, one can expect that the validation of a bug consumes much less resources compared to finding the bug, whereas the proof validation is still resource intensive because the full state-space must be considered. This intuition is supported by the resource-consumption data of the validators in the software-verification competitions. Moreover, in the competition, alarm validation is allocated 10 % (= 1.5 min) of the CPU time of a verification run (15 min), whereas proof validation gets the same CPU time as verification. Both the alarm and proof validation are allocated about half the memory as verification (7 GB).

### 3 Constructing Combinations with COVERTEAM

COVERTEAM [18] is a tool for creating and executing tool combinations. It consists of a language to describe combinations of tools and an execution engine for their execution. Tools, e.g., verifiers, validators, testers, transformers, and their combinations, are called *verification actors* in COVERTEAM. The inputs consumed and outputs produced by the verification actors, e.g., programs, specifications, witnesses, and results, are called *verification artifacts*. Verification artifacts are seen as basic objects, verification actors as basic operations, and tool combinations as compositions of these operations.

Verification actors in COVERTEAM are of two kinds: *atomic* and *composite*. Atomic actors are based on off-the-shelf tool archives. COVERTEAM uses features provided by BENCHEXEC [37] to configure the command line, execute the tool in isolation, enforce resource limits, and process the output produced by the tool. Atomic actors are constructed using the information provided in a YAML configuration file, which specifies the BENCHEXECtool-info module, parameters to pass to the tool, resource limits, and the location to download the tool archive from. Many publicly available tools for automatic verification are supported by COVERTEAM, and their YAML configuration files are available in the COVERTEAM repository [19].

Composite actors are created by combining COVERTEAM actors using the following composition operators: SEQUENCE, PARALLEL, REPEAT, ITE, and PARALLEL-PORTFOLIO. SEQUENCE executes the composed actors sequentially, PARALLEL in parallel, REPEAT repeatedly until the termination condition is satisfied, ITE (*if-then-else*) executes one actor if the provided condition is true, otherwise, the other actor, PARALLEL-PORTFOLIO executes the actors in parallel until one of them finishes with a result that satisfies the success condition, and then terminates all remaining actors [18, 21, 38]. The work in this paper uses SEQUENCE, PARALLEL, ITE, and PARALLEL-PORTFOLIO.

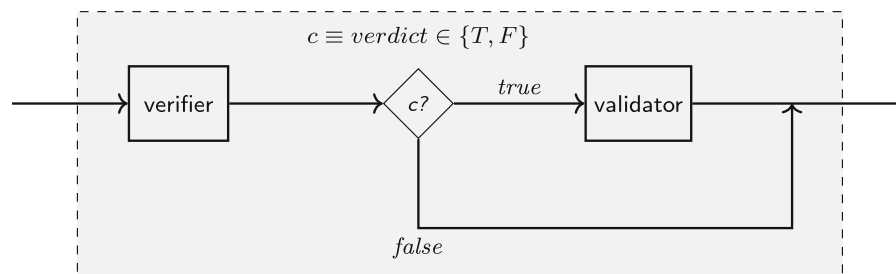


Fig. 3 Construction of a VERIFIER<sup>Val</sup> using COVERTEAM

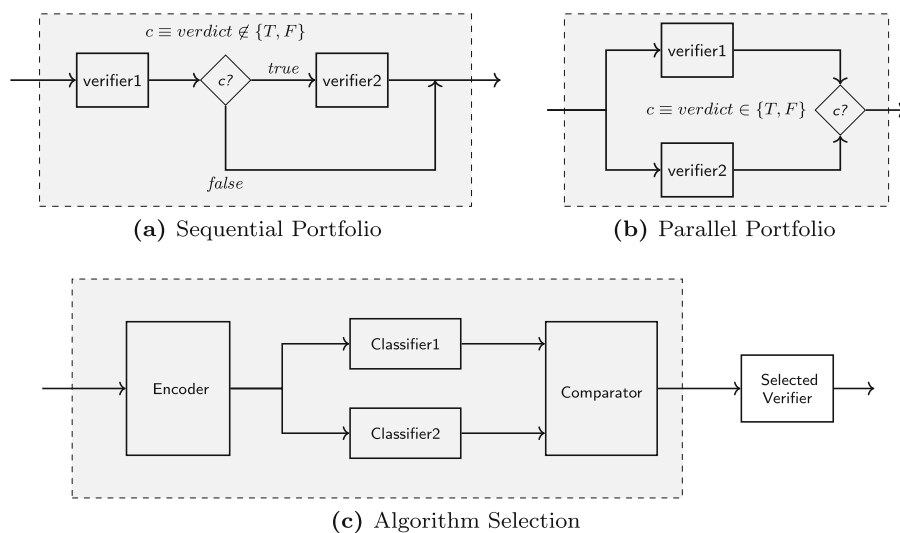
### 3.1 Validating Verifier

Figure 3 shows the construction of a combination  $\text{VERIFIER}^{\text{Val}}$  for a verifier *verifier* and a validator *validator*. This construction uses the COVERITEAM composition operators **SEQUENCE** and **ITE**. The combination first executes the verifier. Then, it checks whether the verifier solved the verification task (i.e., finishes with a verdict *true* or *false*) or not. A verifier might not always succeed; it can finish the execution with an error or be terminated when it runs out of resources. If the verifier solved the verification task, then the combination executes the validator on the result of the verifier, otherwise, the verdict *error* or *unknown* is forwarded. This construction can be generalized to use combinations of validators as *validator*. For our experiments (see Sect. 4.3), we created validating verifiers using a portfolio of validators instead of one validator.

### 3.2 Verifier Based on Sequential Portfolio

Figure 4a shows the construction of a sequential portfolio for two verifiers *verifier1* and *verifier2*. This construction is similar to  $\text{VERIFIER}^{\text{Val}}$ , with the difference that the second actor in this combination is also a verifier instead of a validator. This construction uses the COVERITEAM composition operators **SEQUENCE** and **ITE**. The combination first executes *verifier1*. If the execution of *verifier1* successfully solves the task, then the combination finishes with the result of *verifier1*, otherwise, *verifier2* is executed and the combination finishes with the result of *verifier2*.

This construction can be generalized to create sequential portfolios of arbitrary sizes. For our experiments, we created sequential portfolios of 2, 3, 4, and 8 verifiers.



**Fig. 4** Construction of verifiers based on sequential portfolio, parallel portfolio, and algorithm selection using COVERITEAM (the incoming arrow from the left always represents the verification task; the outgoing arrow on the right always represents the verification result)

### 3.3 Verifier Based on Parallel Portfolio

Figure 4b shows the construction of a parallel portfolio for two verifiers `verifier1` and `verifier2`. This construction uses the COVERTEAM composition operator `PARALLEL-PORTFOLIO`, which combines a set of actors of the same type (verifiers, testers, etc.) based on a success condition. The success condition is defined over the artifacts produced by these actors, and is evaluated whenever an actor finishes its execution. In this combination, both the verifiers are executed concurrently. When one verifier finishes, its verdict is checked for the success condition (i.e.,  $verdict \in \{T, F\}$ ). If the success condition holds, then the combination finishes (terminating all remaining executing verifiers) and returns the verdict, otherwise, the verdict is discarded and the combination waits for the second verifier to finish. If none of the verifiers produces a result that satisfies the success condition, then the combination returns the result of the last verifier. This construction can be generalized to create parallel portfolios of arbitrary sizes. For our experiments, we created parallel portfolios of 2, 3, 4, and 8 verifiers.

### 3.4 Verifier Based on Algorithm Selection

Figure 4c shows the construction of an algorithm selection for two verifiers `verifier1` and `verifier2`. This construction uses the COVERTEAM composition operators `SEQUENCE` and `PARALLEL`, and some COVERTEAM actors for feature encoding, classifiers, and comparator. The combination consists of two parts, the selector to determine an appropriate verifier based on the given verification task and the execution of the selected verifier. In more detail, the combination first executes the feature encoder on the verification task, in which a set of predefined features is extracted and encoded from a given verification task (i.e., certain characteristics that are believed to indicate difficulty for a verifier). The output is passed on to a set of classifiers (`classifier1` and `classifier2`), one for each verifier that is considered for selection. Each classifier predicts the hardness (or difficulty) of the given verification task for the corresponding verifier. The comparator then compares the hardness scores and determines the verifier with the least value, which is predicted to be the most appropriate verifier for the given verification task based on the extracted features, i.e., the verifier that is most likely to solve the task. The last step is to execute only the verifier that was selected (for example, execute `verifier1`, do not execute `verifier2`). This construction can be generalized to create algorithm selections of arbitrary sizes. For our experiments, we created algorithm selections of 2, 3, 4, and 8 verifiers.

**Feature Encoder.** The first component of our construction is the feature encoder. The goal of the feature encoder is to encode the verification task into a meaningful feature-vector ( $FV$ ) representation that can later be used to select a verification tool. Typically, the representation encodes certain features of a program which might correlate with the performance of a verifier such as the occurrence of specific loop patterns [30] or variable types [39, 40].

In this study, we encode verification tasks via a learning-based feature encoder by employing a pre-trained *CSTTransformer* [28]. The *CSTTransformer* first parses a given program  $P$  into a simplified abstract syntax tree (AST) representation. Afterward, a graph-based neural network processes the AST structure and produces a vector representation. The encoding step is learned by pre-training the neural network on selecting various verification tools. While this approach was originally developed to learn a vector representation optimized for a specific verifier combination, the authors have shown that the learned encoder can be effectively reused across many new selection tasks, often outperforming other hand-crafted feature encoders.

**Selection of Verifiers Based on the Individual Difficulty of the Tasks.** One verification tool might be able to solve a given verification task quickly, whereas another tool might fail to solve it even using all given resources. Therefore, to avoid wasting resources on tools that are not well suited for a given task, the algorithm selector aims to predict the difficulty of a task before executing a tool. Then, the tool that is predicted to be the best-suited tool for the task is executed.

Similar to previous work [28], we learn to predict the difficulty of a task with *hardness models* [41]. A hardness model learns to predict the difficulty, also called hardness, of a given task for a specific tool based on the previously computed vector representation of the task. In our case, we define the hardness of a task for a given tool similar to the PAR10 score [42]: It is either the consumed CPU time if the task is solved correctly or ten times the maximal runtime. A low hardness score means that a verifier solves a task correctly in a short amount of time.

Since our hardness score is based on the CPU time consumed by the verifier, the problem of training our hardness model reduces to a regression problem. We address this problem with regression by classification [43] by training multinomial logistic-regression classifiers.

Given a set of hardness models—each assessing the hardness of a verification task for a specific tool—a verification tool is selected for which the task is likely easy, i.e., the respective model outputs the lowest hardness score.

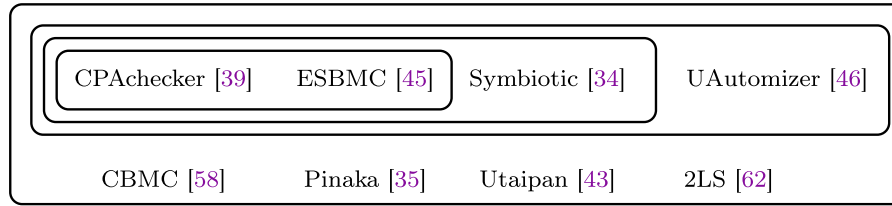
### 3.5 Extensibility

To facilitate future research and the design of novel combinations, we implemented all combination types such that they can be easily configured and extended. Extending a combination with a new verifier requires only an actor definition for that verifier in COVERITEAM. Afterwards, this verifier can be easily added to a sequential or parallel portfolio.

While our algorithm selector can be easily used with all tools employed during our experiments, extending a combination based on algorithm selection with a new verifier requires a bit more effort. However, the task of configuring algorithm selection has been simplified by using hardness models together with a common feature representation.

One can modify the set of verifiers to select from by simply adding or removing individual hardness models. While previous approaches to verifier selection often require training the complete selector from scratch, our combination can be extended by training a single hardness model. A single hardness model can be trained within a few minutes on a modern CPU. The accompanying artifact contains all the training scripts that we used for training our hardness models, a pre-computed dataset of vector representations for SV-COMP 2022, and instructions to train a new model. It is also possible to train and employ custom hardness models based on a custom vector representation. In this case, one needs to replace the feature encoder, which can easily be done as it is a COVERITEAM actor in our construction.

Finally, to integrate a new tool in our algorithm selector, one is only required to run the respective verifier once on (a subset of) the benchmark set. The results then act as training examples.



**Fig. 5** Subsets of 2, 3, 4, and 8 verification tools as used in our combinations

## 4 Experiment Setup

Our goal is to investigate if combinations can yield better results than standalone tools. To achieve this goal, we have chosen the following measures for comparison: number of solved verification tasks, normalized score<sup>1</sup>, and resource consumption. For each combination, we compared the best-performing standalone verifier against the combination using these measures. We derived the following three research questions from our research goal:

- RQ 1.** Can a COVERTEAM-based sequential portfolio of verifiers perform significantly better than standalone tools with respect to
- (a) number of solved verification tasks,
  - (b) normalized score, and
  - (c) resource consumption?
- RQ 2.** Can a COVERTEAM-based parallel portfolio of verifiers perform significantly better than standalone tools with respect to
- (a) number of solved verification tasks,
  - (b) normalized score, and
  - (c) resource consumption?
- RQ 3.** Can a COVERTEAM-based algorithm selection of verifiers perform significantly better than standalone tools with respect to
- (a) number of solved verification tasks,
  - (b) normalized score, and
  - (c) resource consumption?

To address the above research questions, we performed an extensive experimental evaluation. This section explains the setup of our experiment.

### 4.1 Selection of Existing Verifiers

We considered the results of the Competition on Software Verification 2022 [5] for selecting the verification tools for our combinations. We chose the 8 best tools from the REACHSAFETY category, and sorted them according to their scores in SV-COMP 2022. Then we took the top  $n$  tools for a combination of size  $n$ . Figure 5 illustrates the sets of verifiers that we used in different types of combinations.

<sup>1</sup> The benchmark set is partitioned into categories of different sizes. The number of solved verification tasks is biased towards performance in large benchmark sets. Using a normalized score mitigates this bias.

**Exclusions.** We excluded the following verification tools from consideration: VERIABS [49], because its license does not allow us to use it for scientific evaluation, PESCO [50], because it would not contribute to the diversity of technologies in the combinations as it is based on CPACHECKER configurations, GRAVES-CPA [51], for the same reason as PESCO, and CoVeriTeam-VERIFIER-PARALLELPORTFOLIO and CoVeriTeam-VERIFIER-ALGOSELECTION, because they are themselves combinations of verifiers similar to the ones we evaluate in this paper.

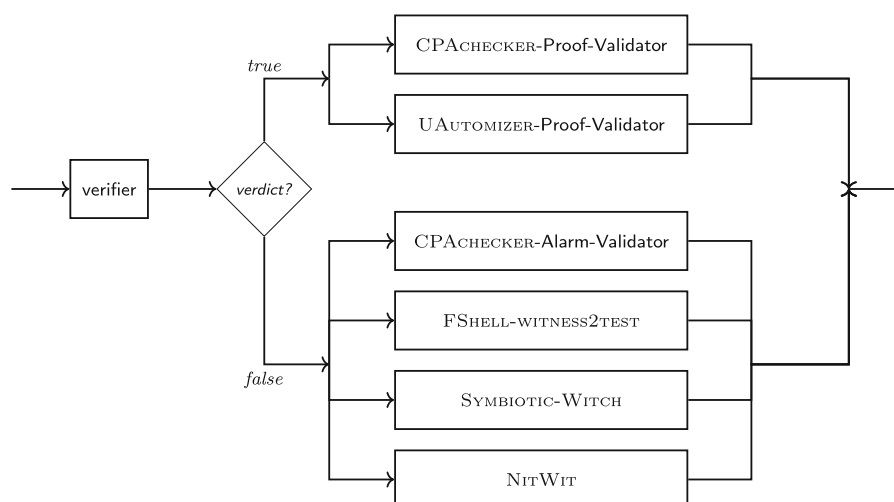
## 4.2 Selection of Existing Validators

We chose the validators also based on the results of the competition on software verification 2022 [5]. We took the validators that were most effective in validating witnesses in the competition as reported in a case study [52]. We took four validators for violation witnesses (*alarm validation*): CPACHECKER-based violation-witness validator [32], FSHELL-WITNESS2TEST [34], SYMBIOTIC-WITCH [53], and NITWIT [54]; and two validators for correctness witnesses (*proof validation*): CPACHECKER- and UAUTOMIZER-based correctness-witness validators [33]. We have excluded METAVAl [55], because it was not adopted to a new rule of SV-COMP 2022 [52].

## 4.3 Construction of VERIFIER<sup>Val</sup> Combinations

Figure 6 shows the construction that validates the results of a verifier. We have used portfolios of validators of different sizes: 2 for validating proofs, and 4 for validating alarms.

The combination first executes the verifier. Then, if the produced *verdict* is *true*, it executes the portfolio of *proof validators*. If the produced *verdict* is *false*, it executes the portfolio of *alarm validators*. If the produced verdict is neither *true* nor *false*, i.e., the verifier did not succeed in its verification effort, then the verdict *error* or *unknown* is simply forwarded. The proof and alarm validators are combined in parallel portfolios, respectively. (The figure shows a simplified presentation of the combination, using a verdict check with



**Fig. 6** Construction of our VERIFIER<sup>Val</sup> combination

three outcomes instead of two ite operators, and omitting the check of success conditions in parallel portfolios.)

We instantiated the VERIFIER<sup>Val</sup> combination for all eight selected verifiers and refer to them as *single verifiers* because they are combined only with validators. We executed the resulting combinations on the chosen benchmark set. The obtained results were also used to train the hardness models used by the algorithm selector.

#### 4.4 Training of the Algorithm Selector

Our selection mechanism is based on the use of hardness models. We trained several hardness models that predict the difficulty of a given verification task for a specific verifier. The algorithm selector then selects the verifier that is most likely to solve a given task, i.e., for which the task is the easiest. In the following, we describe the process for training the individual hardness models used in our evaluation in detail.

**Construction of Training Datasets.** Hardness models learn from prior observations of the verifier's performance to predict the difficulty of future tasks. Therefore, we trained the hardness models on a dataset of verification tasks labeled with the results of the individual verifiers.

We employed a random *subset* of the benchmark set used in SV-COMP 2022 [5] as the training dataset of verification tasks. Therefore, the training dataset partly overlapped with the benchmark set (up to 90%) used in our evaluation. We maintain a fair comparison between algorithm selectors by training them on the same train/test split. To obtain the results of the individual verifiers on the verification tasks, we executed all the single verifiers on the benchmark set. We recorded whether the verifier solves the task *correctly* and the *execution time* in CPU seconds.

Finally, since our hardness models operate on feature-vector representations, we employ our feature encoder to map each verification task to a feature vector. As a result, we obtain  $n$  datasets ( $n$  is the number of verifiers) where each entry maps a feature-vector representation of a verification task to the correctness and execution time of a verifier on that task.

**Training Hardness Models.** The hardness models are trained to predict the hardness of a task for a given verifier. Similar to the PAR10 score, we define the hardness score (h-score) of a given task for a specific verifier as the CPU time if the task can be solved within a certain time limit, or ten times the time limit if the task cannot be solved. For the prediction, we split the range of our h-scores into four intervals<sup>2</sup>:  $[0, 10)$ ,  $[10, 100)$ ,  $[100, 900)$ ,  $[900, 9000]$ . If the verifier solves the task correctly, the hardness model predicts whether solving the task was easy ( $[0, 10)$ ), intermediate ( $[10, 100)$ ) or difficult ( $[100, 900)$ ). In the case that the verifier fails, the hardness model should predict that the task was too hard ( $[900, 9000]$ ).

Motivated by the idea of regression by classification [43], we address this problem by training a multinomial-logistic-regression classifier. Then, for each interval, the classifier predicts the probability that it contains the h-score of the verifier for the given task. Finally, to obtain a predicted hardness score which we can use to select a verifier, we make the following observation for the hardness score:

$$\text{h-score}(x) \leq \sum_{i=0}^k p(\text{h-score}(x) \in [l_i, u_i)) * u_i,$$

where  $x$  is the given verification task,  $k$  is the number of disjoint intervals  $[l_i, u_i)$ . In other words, if we can correctly estimate the probability  $p$  of a hardness score to be included in an

<sup>2</sup> The verification timeout is typically set to 900 s and we found splitting intervals logarithmically works best.

interval  $[l_i, u_i)$ , we can compute an upper bound to the hardness score. We train the logistic-regression classifier to estimate the probability and use the upper bound as the predicted hardness score to select a verifier, i.e., we select the verifier whose hardness score is likely bounded by the smallest constant. We compared our approach with alternative approaches that predict the likelihood of solving an instance [28], solvability and runtime independently [30], or the hardness score via linear-regression models. However, in our experiments, we found that predicting hardness-score intervals leads to the best algorithm-selection performance.

**Selecting a Verifier.** After training, the hardness models predict the difficulty of a given task for a specific verifier. Given a new verification task, the feature encoder is executed followed by the classifiers (hardness models). As a result, we obtain a hardness score for each verifier in our combination. Then, the verifier obtaining the smallest hardness score is selected and executed.

Since the predictions are made independently, our algorithm-selection framework is modular. In other words, we can simply extend or shrink the size of the combination by adding or removing verifiers and their respective hardness models.

#### 4.5 Construction of Portfolio and Selection Combinations

We evaluated twelve verifier combinations. For each sequential portfolio, parallel portfolio, and algorithm selection, we constructed a combination of 2, 3, 4, and 8 verifiers. This gave us four combinations for each combination type. These variants of combinations with different numbers of verifiers allowed us to quantify the influence of the number of verifiers on the performance.

#### 4.6 Resource Allocation

**Resource Allocation for Actors inside VERIFIER<sup>Val</sup>.** Fig. 7 shows the resource limits for the actors inside a VERIFIER<sup>Val</sup> composition. Given the resource limits  $T$  (time) and  $M$

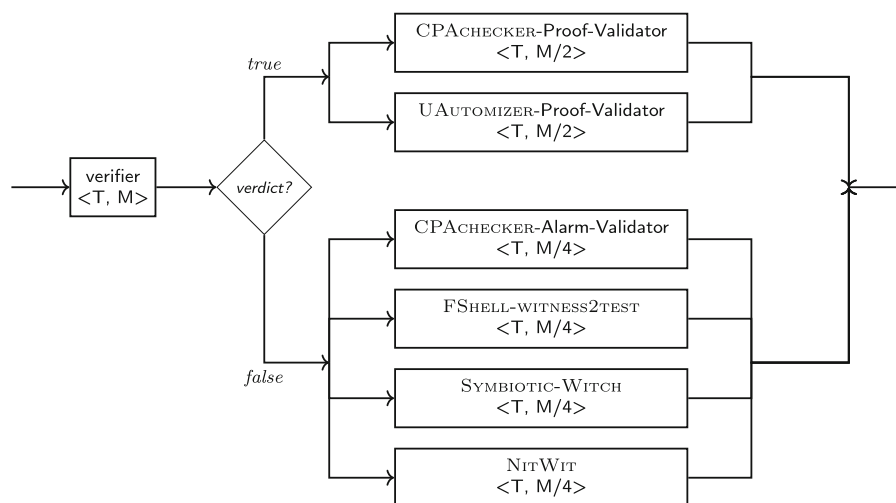


Fig. 7 Resource allocation for the VERIFIER<sup>Val</sup> combination



(memory) for the complete  $\text{VERIFIER}^{\text{Val}}$ , we define the resource limits for the actors in the  $\text{VERIFIER}^{\text{Val}}$  composition as follows:

- Verifier  $\langle T, M \rangle$ : We give the same resource limits to the verifier. It is the first actor to be executed in the combination. As no other tool executes when the verifier is executing, it is fair to allow the verifier to use as much memory as is available. Moreover, as the validation can only start after the verifier finishes execution and produces a result, we give it all the available time.
- Proof-Validator  $\langle T, M/2 \rangle$ : We divide the available memory but pass on the same time limit to the proof-validator. During the validation stage, two proof-validators are running simultaneously. To allow fair distribution of memory we divide the memory limit equally among the two proof-validators. We do not divide the CPU time because this combination is a parallel portfolio of validators and we do not limit CPU time in parallel portfolios.
- Alarm-Validator  $\langle T, M/4 \rangle$ : Analogous to the proof-validators, we divide the available memory but pass on the same time limit to the alarm-validators.

**Resource Allocation for the Combinations of  $\text{VERIFIER}^{\text{Val}}$ .** The resource limits of the combinations of single verifiers (of the form  $\text{VERIFIER}^{\text{Val}}$ ) to a sequential portfolio, parallel portfolio, or algorithm selection are as follows:

- Sequential portfolio: We divide the available CPU time equally among all actors in a sequential portfolio, but allow them to use all the available memory.<sup>3</sup>
- Parallel portfolio: We divide the available memory equally among all actors in a parallel portfolio, but allow them to use all the available CPU time.
- Algorithm selection: We pass on the available resource limits to the feature encoder and the verifier. For classifiers, we divide the memory limit equally and give them a constant time limit of 20 s. We enforce resource limits on the classifiers because classifiers in our experiments take just a couple of seconds to execute. If one of them behaves unexpectedly and consumes too much time, we proceed without waiting for its execution to finish. We select the verifier to execute based on the results of the remaining classifiers.

#### 4.7 Benchmark Selection

We evaluated the tool combinations on a benchmark set from the open-source collection of verification tasks [56]. The benchmark sets for SV-COMP are also selected from this collection. Our benchmark set consisted of all the verification tasks in the category REACHSAFETY used in SV-COMP 2022. It is the largest category, contains 5400 verification tasks, and is the most popular one with 21 participants in SV-COMP 2022. Each verification task consists of a program written in C and a specification. The specification is a safety property describing that an error function should never be called. Thus, we had a total of 5400 verification tasks in our benchmark set. We evaluated our combinations on the version of the benchmark set that was used in SV-COMP 2022 (tag `svcomp22`) [57].

<sup>3</sup> Technical detail: We assign a little bit less memory to the actors of the sequential portfolio because otherwise, if one of them starts consuming more memory than the provided memory limit, then it would make the complete sequential portfolio exceed the memory limit. This would in turn trigger BENCHEXEC to terminate the complete sequential portfolio.

#### 4.8 Execution Environment

Our experiments were executed on machines with the following configuration: one 3.4 GHz CPU (IntelXeon E3-1230 v5) with 8 processing units (virtual cores), 33 GB RAM, and operating system Ubuntu 20.04. Each verification run (execution of one tool or combination on one verification task) was limited to 8 processing units, 15 min of CPU time, and 15 GB memory. This configuration is the same as the configuration used in SV-COMP 2022 allowing us to use the competition results of the standalone tools for comparison.

#### 4.9 Scoring Schema

We report three measures of success for each combination. First, we count the number of results of each kind, i.e., either claims of program correctness or alarms of specification violations for the verification tasks.

Second, we report the scores as per the scoring scheme used in the competition SV-COMP [5]. A verifier is rewarded with score points as follows: 2 score points for each correct proof, 1 score point for each correct alarm, -32 score points for each wrong proof, and -16 score points for each wrong alarm. We have used this scoring scheme because it is accepted in the community as a model of quality. The benchmark set is partitioned into several sub-categories, and we calculate the score for each sub-category and apply normalization as in SV-COMP based on the size of the sub-category. The normalization of scores has been used in SV-COMP [58] for many years and has been established as a standard for judging the quality of results by the verification community.

Although the inclusion of a validation step alleviates the issue of using an unsound verifier to a large extent, it does not eliminate it, because validators could also be unsound and could validate incorrect witnesses. Due to this reason, we still need to use negative scores as well.

The scoring scheme used in our previous work [21] employed the same reward scheme but it did not consider normalization. This resulted in scores being biased towards tools that perform well in the larger sub-categories of the benchmarks. In this article, we report the normalized scores addressing the issue of bias towards the results of the large sub-categories of the benchmarks. We have used the same scripts that were used in SV-COMP to calculate these scores.

#### 4.10 Resource Measurement and Benchmark Execution

We used the state-of-the-art benchmarking framework BENCHEXEC [37] for executing our benchmarks. It executes tools in isolation, reports the resource consumption, and enforces the resource limitations. It provides measurements of the consumption of CPU time, wall time, memory, and CPU energy during the execution of a tool.

#### 4.11 Reporting Results: Tables and Plots

We present a table and two plots for each set of experiments.

**Tables.** We report the normalized score, correctly solved instances of both proofs and alarms, total resource consumption, median resource consumption, and resource consumption per score point for each executed combination.

The tables report the resource consumption only for the correctly solved tasks. It is similar to how the results are presented in the competition reports [5, 59]. Using this approach encourages verifiers to try as hard as possible to solve a verification task without worrying about how it affects the resource consumption. Otherwise, one could, in principle, considerably improve this measure by simply terminating early with the output UNKNOWN.

**Score-Based Quantile Plots.** For each set of experiments, we also present quantile plots based on the normalized score. Each point  $(x, y)$  in these plots represents the score  $x$  accumulated for the executions that finished below the CPU time  $y$ . The CPU time consumption of only those executions that produce a correct result is considered. The time consumption of executions producing incorrect and inconclusive verdicts are not considered. These plots use a linear scale for the CPU time range between 0 and 1 s, and a logarithmic scale for 1 s to 1000 s.

Interpretation: The higher the score of a tool, the farther on the right its plot goes. As our scoring scheme penalizes incorrect results, the abscissa of the starting point of each plot line is the total penalty a tool has received. The more unsound a tool is, the farther on the left its plot graph starts. The length of the projection of the plot graph on the horizontal axis loosely corresponds to the total number of correctly solved tasks (because 2 points are awarded for a correct proof, and 1 point for a correct alarm). The height of a plot represents the maximum time required by the corresponding tool to correctly solve a verification task. The area under the graph loosely corresponds to the total time taken by the tool for the executions that resulted in correct results. In essence: the plot graph of a sound, effective, and efficient tool would start at zero on the x-axis, go far towards the right, and remain low. More details about these plots are given in [58].

**Parallel-Coordinates Plots.** In addition to the tables and quantile plots, we present parallel-coordinates plots for showing the resources consumed per score point. Parallel-coordinates plots are used to display multivariate data points, where each variable gets its own axis and each graph represents one data point. They provide a visual aid to compare many variables and see the relation between them.

Another possibility to show the resource consumption per score point was to use spider charts (also known as radar or web charts). In a spider chart, linear differences in values of a variable scale to a quadratic change in the area, which may give an incorrect impression to a viewer. Therefore, we chose to use parallel-coordinate plots instead of spider charts.

The plots show resource consumption per score point, as well as the number of unsolved tasks per score point. The lower the plot graph remains the better it is.

## 5 Evaluation Results

### 5.1 Results for VERIFIER<sup>Val</sup> Standalone Compositions

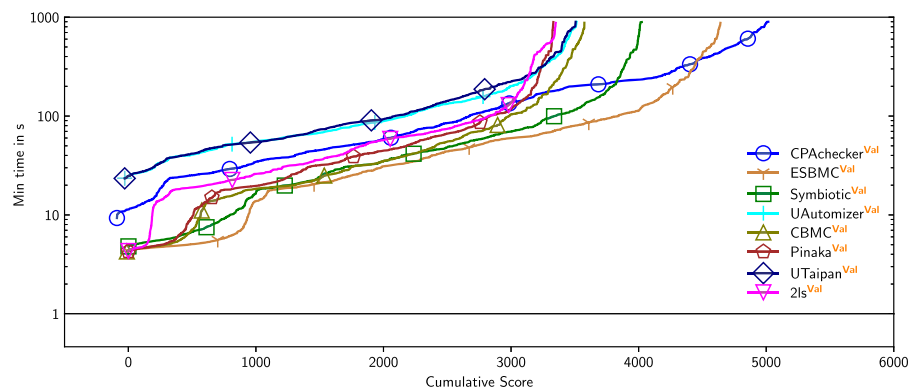
Table 1 shows the summary of results for executing the VERIFIER<sup>Val</sup> compositions, that is, combinations of an existing verifier with a validator portfolio as a standalone composition (referred to as *single verifiers*). The scores and ranking are roughly comparable to the results of SV-COMP 2022.<sup>4</sup> Introducing the validation step decreased the scores by about 10 %. Figure 8 shows the score-based quantile plot of the results, and Fig. 9 shows the parallel-coordinates plot for unsolved tasks and resource consumption per score point.

<sup>4</sup> <https://sv-comp.sosy-lab.org/2022/results/results-verified>.

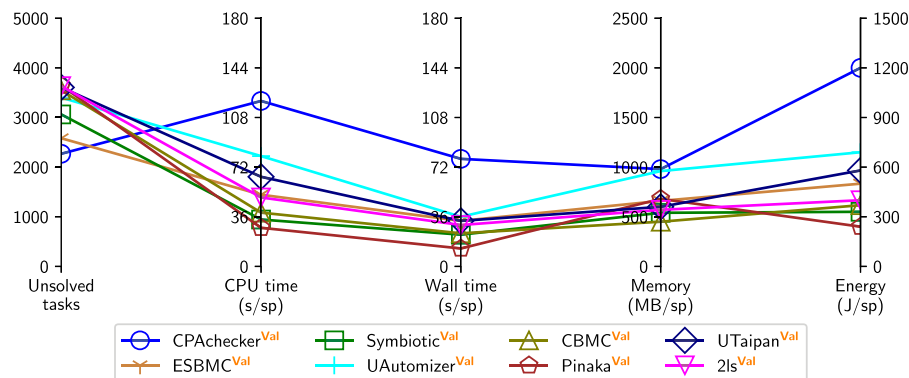
**Table 1** Single verifiers: comparison of results of VERIFIER Val (ordering of columns is based on the score of the underlying verification tools in SY-COMP 2022)

Verifier	CPACHECKER	Val	ESBMC	Val	SYMBIOTIC	Val	UAUTOMIZER	Val	CBMC	Val	PINAKA	Val	UTAIPAN	Val	2LS	Val
Score, normalized	<b>5016</b>		4 641		4 025		3 513		3 575		3 332		3 506		3 350	
Correct results	<b>3 129</b>		2 818		2 338		1 997		1 852		1 755		1 792		1 768	
Correct proofs	<b>1 574</b>		1 453		8 53		1 215		743		845		1 194		1 186	
Correct alarms	<b>1 555</b>		1 365		1 485		782		1 109		910		598		582	
Wrong results	2		<b>0</b>		<b>0</b>		2		1		<b>0</b>		2		<b>0</b>	
Wrong proofs	<b>0</b>		<b>0</b>		<b>0</b>		2		<b>0</b>		<b>0</b>		2		<b>0</b>	
Wrong alarms	2		<b>0</b>		<b>0</b>		<b>0</b>		1		<b>0</b>		<b>0</b>		<b>0</b>	
Total resource consumption for correct results																
CPU time (h)	170		67		38		78		39		<b>26</b>		63		46	
Wall time (h)	110		42		26		35		24		<b>12</b>		32		28	
Memory (GB)	4 900		3 000		2 200		3 400		<b>1 600</b>		2 300		2 100		1 900	
CPU Cpuenergy (KJ)	6 200		2 300		1 300		2 400		1 300		<b>790</b>		2 000		1 300	
Median resource consumption for correct results																
CPU time (s)	150		45		24		86		<b>22</b>		31		85		54	
Wall time (s)	86		13		13		27		<b>8.0</b>		11		25		15	
Memory (MB)	1 100		680		<b>530</b>		820		550		610		780		710	
CPU Cpuenergy (J)	1 500		310		230		640		<b>170</b>		250		620		390	
Resource consumption of correct results per score point																
CPU time (s/sp)	120		52		34		80		39		<b>28</b>		65		50	
Wall time (s/sp)	78		33		23		36		24		<b>13</b>		33		30	
Memory (MB/sp)	980		660		540		960		<b>450</b>		680		600		570	
Cpuenergy (J/sp)	1 200		500		330		690		370		<b>240</b>		580		400	

## Formal Methods in System Design



**Fig. 8** Single verifiers: Score-based quantile plot for results of VERIFIER<sup>Val</sup> combinations



**Fig. 9** Single verifiers: Unsolved tasks and resource consumption per score point of VERIFIER<sup>Val</sup> combinations

## 5.2 RQ 1: Evaluation of Sequential-Portfolio Verifiers

We now present the results of the sequential-portfolio verifiers against the standalone VERIFIER<sup>Val</sup> combination with the highest score: CPACHECKER<sup>Val</sup>.

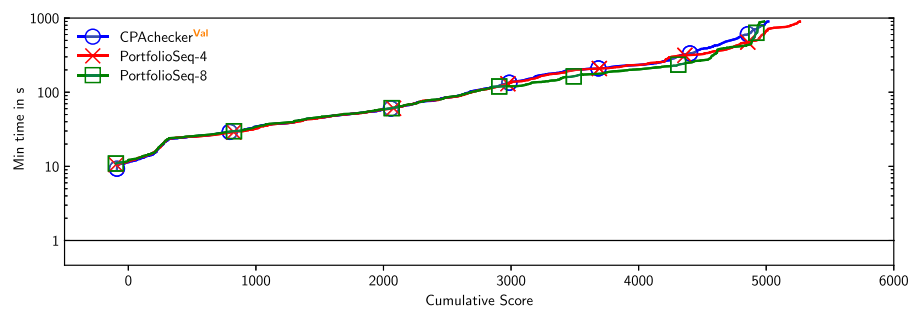
Table 2 shows the summary of results for the sequential portfolio. Three of our sequential portfolios achieve a better score than CPACHECKER<sup>Val</sup>. The portfolio with 8 tools performs worst, which is expected because the amount of time allocated to each verifier decreases as we increase the size of the portfolio. As a result, verifiers cannot solve hard tasks that take long to solve. The table also shows that the portfolios require more resources to solve the tasks. This is a side effect of the sequential portfolio, as all the resources consumed by unsuccessful attempts by the verifiers in a sequence are still counted towards the overall resource consumption.<sup>5</sup>

Figure 10 shows the quantile plot of normalized scores for the best and worst performing sequential portfolios, and CPACHECKER<sup>Val</sup>. All graphs start from the same abscissa because all of them have the same number of incorrect results (a negligible value of 2). The sequential portfolio of 4 tools goes farthest to the right because it has the highest score. Figure 11 shows

<sup>5</sup> This may change with a change in the order of tools in the sequence. One could try to come up with an optimum order by analyzing the results of the standalone CPACHECKER<sup>Val</sup>. But we kept our approach simple and put the tools in the order of the SV-COMP scores.

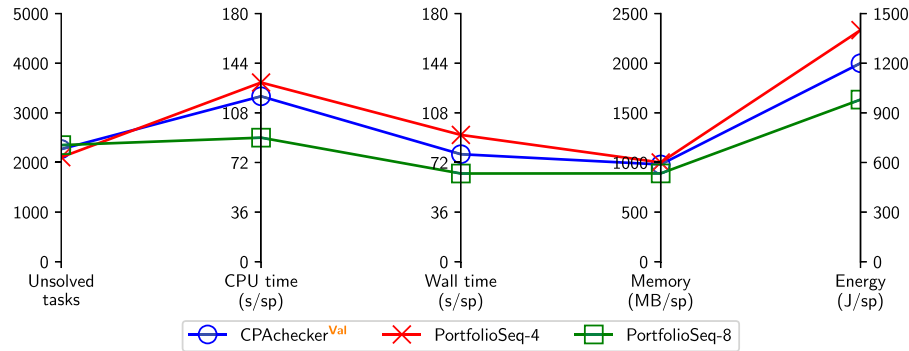
**Table 2** Sequential portfolios: comparison different sizes with CPACHECKER<sup>Val</sup>

Verifier	CPACHECKER <sup>Val</sup>	Sequential Portfolio of			
		2	3	4	8
Score, normalized	5 016	5 224	5 230	<b>5 265</b>	4 984
Correct results	3 129	3 239	<b>3 298</b>	3 292	3 048
Correct proofs	1 574	<b>1 621</b>	1 587	1 601	1 385
Correct alarms	1 555	1 618	<b>1 711</b>	1 691	1 663
Wrong results	<b>2</b>	<b>2</b>	<b>2</b>	<b>2</b>	<b>2</b>
Wrong proofs	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>
Wrong alarms	<b>2</b>	<b>2</b>	<b>2</b>	<b>2</b>	<b>2</b>
Total resource consumption for correct results					
CPU time(h)	170	190	190	190	<b>120</b>
Wall time (h)	110	130	130	130	<b>88</b>
Memory (GB)	4 900	5 400	5 600	5 300	<b>4 400</b>
CPU Energy (KJ)	6 200	7 000	7 200	7 300	<b>4 900</b>
Median resource consumption for correct results					
CPU time(s)	150	160	170	160	<b>120</b>
Wall time (s)	86	96	110	110	<b>77</b>
Memory (MB)	1 100	1 100	1 100	1 000	<b>950</b>
CPU Energy (J)	1 500	1 600	1 800	1 700	<b>1 300</b>
Resource consumption of correct results per score point					
CPU time (s/sp)	120	130	130	130	<b>90</b>
Wall time (s/sp)	78	88	93	92	<b>64</b>
Memory (MB/sp)	980	1 000	1 100	1 000	<b>890</b>
CPU Energy (J/sp)	1 200	1 300	1 400	1 400	<b>980</b>

**Fig. 10** Sequential portfolios: Score-based quantile plot comparing CPACHECKER<sup>Val</sup>, the best and the worst-performing sequential portfolios (SeqPortfolio-4 and SeqPortfolio-8, respectively)

that sequential portfolios can be less resource-efficient in comparison to CPACHECKER<sup>Val</sup>. In general, as we increase the size of the sequential portfolio and thereby increase its effectiveness, we also decrease its resource efficiency. The trend of increased effectiveness is visible for all sequential portfolios, except for the sequential portfolio of 8 tools. In this case, the verifiers in the portfolio of 8 tools are not provided with enough resources to solve some of

Formal Methods in System Design



**Fig. 11** Sequential portfolios: Unsolved tasks and resource consumption per score point of CPACHECKER<sup>Val</sup>, the best and the worst-performing sequential portfolios (SeqPortfolio-4 and SeqPortfolio-8, respectively)

the given tasks, which reduces the effectiveness of the portfolio but also increases its resource efficiency for computing correct results.

This is also visible in the plot graph for the sequential portfolio of 8 tools in Fig. 10. Here the plot graph goes most toward the right in three steps and after that its slope increases considerably, showing that the later verifiers did not contribute much to the number of correctly solved tasks.

**Difference to previous results [21].** A key observation of our previous work [21] is that portfolios are negatively affected by wrong results produced by unsound tools. As a consequence, sequential portfolios often perform worse overall even though they can achieve a higher number of correct results. In this work, we mitigate the impact of wrong results by employing validating verifiers. As a consequence, our sequential portfolios do not produce more incorrect results than the best standalone VERIFIER<sup>Val</sup> composition CPACHECKER<sup>Val</sup>. As a side effect, sequential portfolios are now able to achieve a higher score.

### 5.3 RQ 2: Evaluation of Parallel-Portfolio Verifiers

We now present the results for the parallel portfolio. Table 3 shows the summary of results for the parallel portfolios. Similar to the sequential portfolio, the parallel portfolio solves a higher number of verification tasks in comparison to CPACHECKER<sup>Val</sup> and thereby also achieves a higher score. Initially, the score increases with the size of a parallel portfolio. However, as the size becomes too large to give each verifier reasonable resources, the performance starts to decrease.

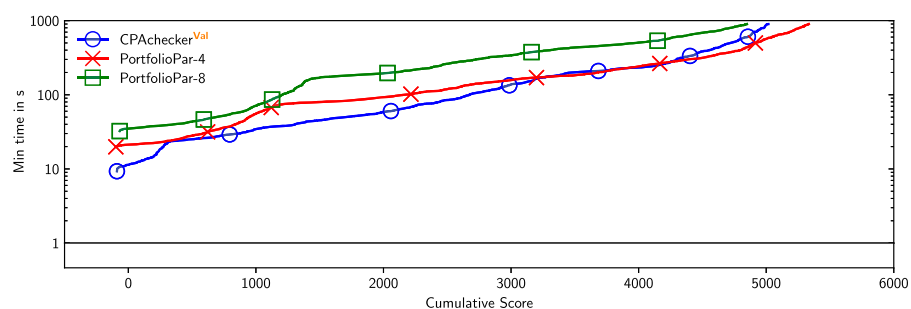
Figure 12 shows the quantile plot of normalized scores for the best and worst-performing parallel portfolios, and CPACHECKER<sup>Val</sup>. All graphs start from nearly the same abscissa because all of them have the very low number of incorrect results. The portfolio of size 4 goes farthest to the right because it has the highest score.

Figure 13 shows that the best-performing parallel portfolio performs better than CPACHECKER<sup>Val</sup> in terms of resource efficiency except for memory consumption. Higher memory consumption is expected as several tools are running in parallel. A lower wall-time is expected for the same reason. The reduction in CPU time is interesting, which we attribute to the diversity of the benchmark set: some tasks are simple for one tool but harder for another, and vice versa.

The resource consumption of the worst-performing parallel portfolio is worse than for CPACHECKER<sup>Val</sup>. Portfolios of large sizes do not provide enough resources for any verifier

**Table 3** Parallel portfolios: Comparison of different size with CPACHECKER

Verifier	CPACHECKER <sup>Val</sup>	Parallel Portfolio of			
		2	3	4	8
Score, normalized	5 016	5 300	5 309	<b>5 334</b>	4 849
Correct results	3 129	3 311	<b>3 384</b>	3 351	2 977
Correct proofs	1 574	<b>1 620</b>	1 596	1 614	1 322
Correct alarms	1 555	1 691	<b>1 788</b>	1 737	1 655
Wrong results	2	2	2	2	<b>1</b>
Wrong proofs	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>
Wrong alarms	2	2	2	2	<b>1</b>
Total resource consumption for correct results					
CPU time(h)	170	<b>150</b>	160	180	230
Wall time (h)	110	56	43	36	<b>32</b>
Memory (GB)	<b>4 900</b>	7 200	8 400	9 500	11 000
CPU Energy (KJ)	6 200	4 500	4 100	<b>4 000</b>	<b>4 000</b>
Median resource consumption for correct results					
CPU time(s)	150	91	<b>83</b>	130	200
Wall time (s)	86	17	<b>16</b>	21	27
Memory (MB)	<b>1 100</b>	1 400	1 500	1 800	3 100
CPU Energy (J)	1 500	550	<b>530</b>	740	1 000
Resource consumption of correct results per score point					
CPU time (s/sp)	120	<b>100</b>	110	120	170
Wall time (s/sp)	78	38	29	<b>24</b>	<b>24</b>
Memory (MB/sp)	<b>980</b>	1 400	1 600	1 800	2 200
CPU Energy (J/sp)	1 200	840	770	<b>750</b>	830

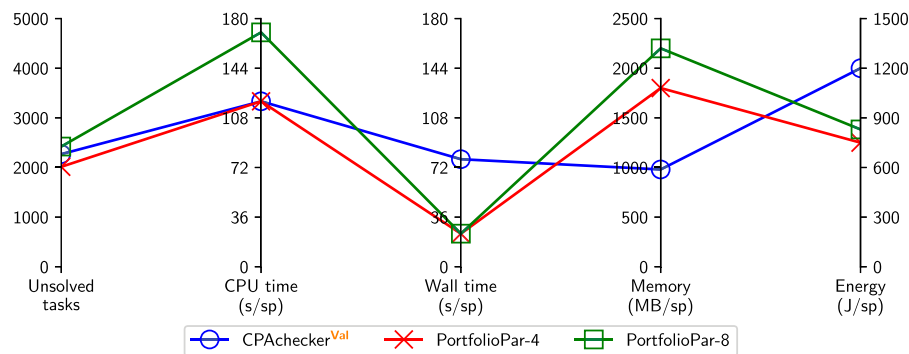
**Fig. 12** Parallel portfolios: score-based quantile plot comparing CPACHECKER<sup>Val</sup>, the best and the worst-performing parallel portfolios (ParPortfolio-4 and ParPortfolio-8, respectively)

to compute a result. As a result, only those verification tasks get solved that are *easy* for at least one verifier in the portfolio. Nonetheless, since several tools are running in parallel, the CPU time and memory are still accounted for even this short execution time. We can see that the wall time for the parallel portfolio of size 8 is the least.

**Difference to previous results [21].** In addition to producing a high number of wrong results, unsound tools often produce wrong results quickly [21]. For parallel portfolios, this impacts



Formal Methods in System Design



**Fig. 13** Parallel portfolios: Unsolved tasks and resource consumption per score point of CPAchecker<sup>Val</sup>, the best and the worst-performing parallel portfolios (ParPortfolio-4 and ParPortfolio-8, respectively)

the performance negatively as wrong results that are produced quickly are selected before a correct result can be computed. Therefore, similar to sequential portfolios, parallel portfolios based on validating verifiers did not produce a higher number of wrong results than the single best tool CPAchecker<sup>Val</sup>. (Interestingly, the parallel portfolio of size 8 produces even fewer incorrect results. This was due to timeout.)

### 5.4 RQ 3: Evaluation of Algorithm-Selection Verifiers

Table 4 shows the summary of results for algorithm selection: There is a clear trend towards better results with more verifiers. This is expected because our selector has more options to choose from, including verifiers that are more effective for some tasks. Also, algorithm-selection-based verifiers do not need to share resources between verifiers. Therefore, they can benefit from multiple verifiers without wasting resources on unsuccessful verification attempts. The number of wrong results is, as expected, relatively low.

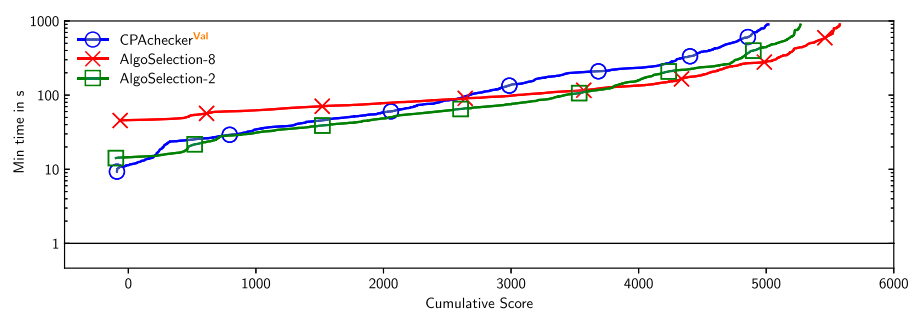
In Fig. 14, all plots start from around similar scores but at different times. Algorithm-selection-based verifiers have a higher startup time than the standalone CPAchecker<sup>Val</sup> because of the overhead of the selection process. This difference in CPU time consumption is much more pronounced for the verification tasks that were solved quickly by the chosen verifier, but as the verifier starts dominating the CPU time consumption on more difficult tasks, this overhead of selection starts to pay off. We can observe that CPAchecker<sup>Val</sup> performs initially better with respect to CPU time, but after around the midpoint, algorithm selection starts to be more efficient.

Figure 15 shows that algorithm selection is also more resource-efficient than CPAchecker<sup>Val</sup> except for peak memory consumption. By design, the algorithm selector aims to predict the fastest verifier that solves the given task successfully.

There is a linear increase in CPU time and memory overhead with the number of choices the algorithm selector is given. We attribute this to using an off-the-shelf combination (see Fig. 4c) instead of an integrated one for the selection algorithm. Our construction allows adding a verifier just by adding the classifier based on hardness models. Increasing the number of tools for selection also increases the number of *classifiers* called. And since each of them is used as an off-the-shelf tool, the overhead of starting the classifier is added to the resource consumption. This explains the relatively high startup time for the algorithm selection with size 8. The quantile plot for peak memory consumption (Fig. 16) also shows

**Table 4** Algorithm Selections: Comparison of different sizes with CPACHECKER

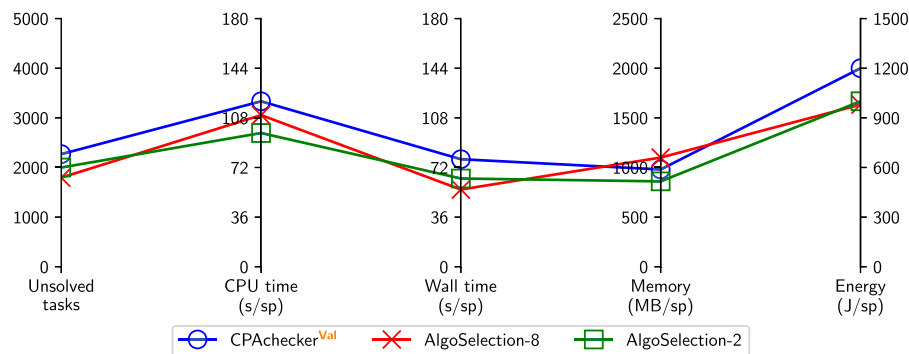
Verifier	CPACHECKER <sup>Val</sup>	Algorithm Selection of			
		2	3	4	8
Score, normalized	5 016	5 268	5 420	5 563	<b>5 577</b>
Correct results	3 129	3 399	3 511	3 596	<b>3 604</b>
Correct proofs	1 574	1 632	1 647	1 716	<b>1 726</b>
Correct alarms	1 555	1 767	1 864	1 880	<b>1 878</b>
Wrong results	2	2	<b>1</b>	<b>1</b>	<b>1</b>
Wrong proofs	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>
Wrong alarms	2	2	<b>1</b>	<b>1</b>	<b>1</b>
Total resource consumption for correct results					
CPU time(h)	170	<b>140</b>	<b>140</b>	150	170
Wall time (h)	110	94	92	92	<b>86</b>
Memory (GB)	4 900	<b>4 500</b>	4 600	5 000	6 000
CPU Energy (KJ)	6 200	<b>5 200</b>	<b>5 200</b>	5 500	5 400
Median resource consumption for correct results					
CPU time(s)	150	77	<b>74</b>	83	110
Wall time (s)	86	<b>30</b>	33	33	32
Memory (MB)	1 100	920	<b>850</b>	<b>850</b>	1 200
CPU Energy (J)	1 500	640	<b>630</b>	690	770
Resource consumption of correct results per score point					
CPU time (s/sp)	120	97	<b>94</b>	99	110
Wall time (s/sp)	78	64	61	60	<b>56</b>
Memory (MB/sp)	980	860	<b>840</b>	890	1 100
CPU Energy (J/sp)	1 200	1 000	<b>960</b>	980	980

**Fig. 14** Algorithm selections: Score-based quantile plot comparing CPACHECKER<sup>Val</sup>, the best and the worst performing algorithm selection based VERIFIER<sup>Val</sup> (AlgoSelection-8 and AlgoSelection-2, respectively)

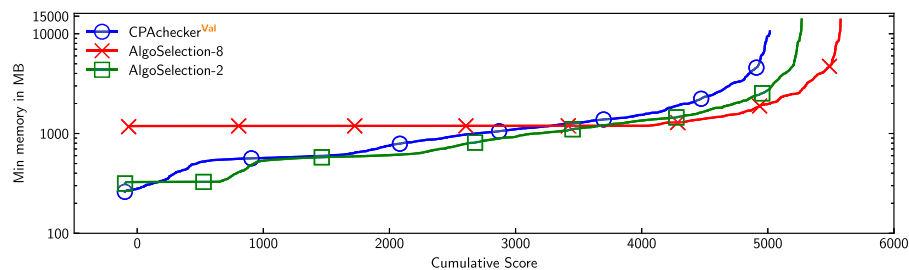
this: it starts with higher memory consumption relative to CPACHECKER<sup>Val</sup> but then the line remains horizontal for most of the graph.<sup>6</sup>

<sup>6</sup> The version of COVERITEAM used in our previous work [21] did not execute the classifiers concurrently so the peak memory consumption was not high. The (conceptually) parallel composition was implemented by executing the tools one after another and then combining the results. The newer version of COVERITEAM executes tools concurrently. Due to this reason, we did not notice the increase in memory consumption in the results for algorithm selection of size 8 in our previous work [21].

## Formal Methods in System Design



**Fig. 15** Algorithm selections: Unsolved tasks and resource consumption per score point CPAchecker<sup>Val</sup>, the best and the worst performing algorithm selection based VERIFIER<sup>Val</sup> (AlgoSelection-8 and AlgoSelection-2, respectively)



**Fig. 16** Algorithm selections: Score-based quantile plot for memory consumption comparing CPAchecker<sup>Val</sup>, the best and the worst performing algorithm selection based VERIFIER<sup>Val</sup> (AlgoSelection-8 and AlgoSelection-2, respectively)

**Difference to Previous Results. [21]** . The main results for the algorithm-selection-based verifiers confirm the results presented in our previous work [21]. There is one improvement though: In our previous work, we employed a simpler algorithm selector that did not consider the resource consumption of individual verifiers during selection. In contrast, our new algorithm selector prioritizes verifiers that solve a verification task not only correctly but also quickly. This change in design is visible in our experimental results. The algorithm-selection-based verifiers consume significantly fewer resources, which is visible for both CPU and wall time.

### 5.5 Discussion

Our experiments show that each combination can on average perform better than any standalone VERIFIER<sup>Val</sup> in terms of correctly solved tasks. This is also true for the normalized scores.

We were expecting that portfolios would be less effective in comparison to the standalone tools because of higher resource consumption. In particular, we were expecting that they would be unable to solve hard tasks as less resources would be allocated to each participating tool. However, the experimental data demonstrate the opposite. A portfolio would be unable to solve tasks that are hard for each tool in the portfolio. Our benchmark set had few such tasks. But for most of the tasks that were hard for one tool, there was some

other tool in the portfolio that could solve it in the allocated time. This was especially pronounced in the parallel portfolio.

The outcome regarding resource consumption is in agreement with our expectations. In comparison to the best performing standalone VERIFIER<sup>Val</sup>, sequential portfolios require more time but perform better with respect to memory consumption. Whereas, parallel portfolios perform better with respect to wall-time but have higher memory consumption. It seems that the portfolios are more energy-efficient when more cores are being used.

Our algorithm selection is based on a model trained using machine learning. The training penalizes the tools that produce more incorrect results and also considers the resource consumption in terms of CPU time. In comparison to both portfolios, the verifier based on algorithm selection solved more tasks.

Our verifier combinations can be constructed by simply selecting tools that perform well in a comparative evaluation, such as the Competition on Software Verification. We found that it leads to successful combinations for all evaluated combination types. Nevertheless, the combinations can be further fine-tuned to achieve even better results.

The portfolio combinations are easy to construct and can perform well if the set of tools to combine is diverse (different strengths). Also, the portfolios should not be too large unless we are willing to increase the resources. Training the algorithm selector requires more preliminary work, but with limited resources and enough choices (number of tools), the selection-based verifier becomes more effective.

**Difference to Previous Results [21].** A key observation of our previous work [21] is that portfolios prefer fast results and unsound tools may produce wrong results quickly. Therefore, the soundness of portfolios can be affected by fast unsound tools. Since this could partially be addressed by the execution order in sequential portfolios, this effect was more pronounced in parallel portfolios than in sequential portfolios. To mitigate this problem in general, we suggested to introduce a validation step. In this work, we adopted the proposed mitigation strategy and updated our experiment setup to include validation. Our results show that it achieves the intended effect as the number of incorrect results decreases significantly.

**Correct vs. Wrong Results.** Interestingly, and in line with our previous work [21], we can observe a tradeoff between correct results and wrong results. By combining tools, we can easily increase the number of solved tasks but we also risk increasing the number of wrong results. The validation step introduced in this work effectively reduces the number of wrong results but also limits the number of tasks that can be correctly verified by the individual verifiers.

In the end, our works present three types of easy-to-deploy combinations of software verifiers, both with and without validation. A user can choose to include validation if the user prefers to ensure correct results, otherwise, the user can choose to omit the validation step. Furthermore, we are convinced that investigating combination types with a better tradeoff of correct vs. wrong results is an exciting research direction.

**Type of Verification Tools.** COVERITEAM supports only automated verification tools, therefore, all the tools in a combination must be automated. For our combinations, we have considered tools based on their performance in SV-COMP. These happen to be model checkers, instead of deductive software verifiers like DAFNY [60], FRAMA-C [61], KEY [62], VERCORS [63], VERIFAST [64], etc.

Any automated verifier that runs on a Linux system can be integrated in COVERITEAM. To integrate a new tool, one needs to create a self-contained archive that is available online, write a tool-info module to assemble the command line and process the output, an actor definition for COVERITEAM to orchestrate the execution of the tool and process artifacts, and have a corresponding actor in COVERITEAM. COVERITEAM already contains a library of

verification actors and actor definitions for many publicly available verification tools. More details on how to integrate a new tool are available in the article on COVERTEAM [18].

**Comparison Between Different Types of Combinations.** Our evaluation compares different types of combinations with the best-performing standalone tool. Although it is enticing, interesting, and valuable to investigate how different combinations perform in comparison to each other, this paper focuses only on how they compare to standalone tools.

A comparative evaluation of different combinations would require a different experiment setup. Different factors influence the performance of each of these combinations, e.g., the position of a verifier in a sequential portfolio, the training set for the learning-based selector, set of chosen verifiers in portfolios. Our experiment setup uses the top  $n$  verifiers based on a list and used them in these combinations. This setup is inadequate to conclude how these combinations would perform against each other. We leave the optimal configuration of our combination types and the comparison between combinations open for future work.

## 6 Threats to Validity

### 6.1 External Validity

**Selection of verifiers.** The effectivity of a combination of tools depends on the effectivity of its parts. Therefore, the performance of a concrete instantiation of our tool combinations is influenced by the selected tools and their configuration, and our results might not generalize to other selections of tools. We have selected the eight most powerful verification tools of the category REACHSAFETY based on the results of the competition SV-COMP and executed them in their original configuration as submitted to the competition. Our procedure to select the verifiers to include in the combination is described in Sect. 4.1.

**Applicability to other verification tasks.** Our evaluation results are based on experiments with a given benchmark set. While we have evaluated our tool combinations on programs taken from the largest and most diverse set of publicly available verification tasks for C programs, the performance of the evaluated combinations may be different on other sets of verification tasks. The selection of the benchmark set is described in Sect. 4.7.

**Training of the algorithm selector** The choice of the benchmark set also impacts the training of our algorithm selector. Training a learning-based selector requires a large and diverse set of verification tasks. Each task has to be labeled with the execution results of each verifier used in our combinations. The used benchmarks repository [56, 57] was created and improved by the verification community over many years. We are not aware of any other benchmark set of verification tasks that is as diverse and of the same quality as this one. As a result, we had to train our algorithm selector on the same dataset that we later used for benchmarking the tool combinations. Therefore, our evaluation shows that algorithm selection improves the performance of verification on the given benchmark set and the selector might generalize only to benchmark sets with similarly distributed verification tasks. For a fair comparison, we (1) restricted the training to linear models, which are known to generalize well, (2) trained only on a random subset of the benchmark set, and (3) cross-validated our model over multiple benchmark splits. The variance of selection performance between different splits was less than 1 %. Therefore, the performance of our trained algorithm selector is likely independent of the random subset selected for training.

**Design of the algorithm selector** The evaluation of algorithm selection is also dependent on the chosen selection technique. Choosing alternative selection methods, e.g., based on hand-

crafted rules, might impact the evaluation. However, the design of hand-crafted methods is not straightforward, it might require expert knowledge about the tool implementations of the components. This design process might in addition be biased in favor of certain tool combinations, which could also impact the experimental results.

**Experiment environment.** The setup of our experiments is influenced by SV-COMP: benchmark set, tool selection, the configuration in which tools are used, execution environment, and resource limitations. On the one hand, it gives us the benefit that our results could be compared with the evaluation of many publicly available well-known verification tools, on the other hand, it affects the generalizability of our results. However, over the last decade, the setup used by SV-COMP has become standard in the verification community for the evaluation of verification tools and this was the best choice available to us. Also, using the SV-COMP setup allows us to compare the results of our combinations with the results of the standalone tools from SV-COMP 2022.

**Sequential portfolios.** The order of the verifiers in sequential portfolios may impact its performance. We ordered the verifiers in sequence according to their performance in SV-COMP 2022, that is, the best performing tool is executed first, and so on. Changing the order of the tools might change the results concerning resource consumption. We noted in our previous work [21] that changing the order of verifiers can impact the soundness of the combination. This was happening if an unsound and fast verifier was put early in the sequence. We now mitigate this issue by validating the results produced by the verifiers.

**Validation step.** The results of our evaluation are dependent on the quality of employed validators and witnesses produced by the verifiers. We have used the validators participating in SV-COMP 2022. The selection of the validators to include in the combination is described in Sect. 4.2. We used a portfolio of validators because no standalone validator could effectively validate the results produced by most of the verifiers. Each validator was more effective for some subset of verifiers. Our results could change with a different selection of validators or a different quality of witnesses produced by the verifiers.

## 6.2 Internal Validity

**Experiment setup.** We have used the same verifier archives, benchmark set, benchmarking framework, resource limits, and infrastructure to execute our experiments as was used for SV-COMP 2022. The unchanged execution setup ensures that there are no unintended side effects in our experiments. Also, since we did not change any component of the verifiers and executed them with the same parameters inside the combination and when executed as a standalone verifier, we exclude the possibility that we could have used the verifiers in a sub-optimal way.

**Memory and time overhead.** COVERITEAM induces an overhead of about 0.8 s for each actor in the combination and around 44 MB memory overhead [18]. It is possible to reduce this overhead by using shell scripts, but we decided in favor of using COVERITEAM for composing tools because it supports modular design. This is especially pronounced in our algorithm-selector combination. We could have saved a few seconds if we were using a monolithic algorithm selector instead of composing one.

**Measurement and control of resources.** We have used BENCHEXEC [37] to measure CPU time and memory consumption, and to enforce the resource limits. Since BENCHEXEC is based on the modern features of the Linux kernel and thus the most accurate measurement technology, we eliminate the measurement-related confounding factors in our evaluation according to the state of the art.

**Distribution of CPU cores.** We rely on the operating system for a fair distribution of CPU usage during the execution of parallel portfolios. In general, there might be a tool that uses multi-processing and as a result would consume more CPU time as compared to a tool using only one process. In such a portfolio, where some tools heavily use multi-processing and others use only one process, the actors using multi-processing would unfairly consume more CPU time and deliver more results. COVERTeam can only control the limits. A user of a portfolio can limit the CPU time available to each tool, resulting in the termination of tools that consumed the allocated CPU time, but whether CPU time is consumed sooner or later by a process is decided by the operating system. In our experiments with parallel portfolios, we allow tools to use the CPU time left by tools that terminated earlier.

### 6.3 Construct Validity

Our experiments are designed to assess whether combinations of verifiers can improve effectiveness and efficiency compared to standalone verifiers. The measures that we use to quantify the quality are the community-agreed scoring schema, the number of solved tasks, and the resource measures memory, CPU time, wall time, and CPU energy. These measures are all standards accepted by the verification community and have also been used in the competition on software verification for many years.

## 7 Related work

**Combination Types Used in Software Verification.** Combining verifiers to increase the verification performance is a well-established technique in the domain of software verification [14, 23, 24, 27, 29, 50, 51, 65–68]. The top three winning entries of the software-verification competition SV-COMP 2022 all combine various verification techniques to achieve their performance [5]. CPACHECKER [29] combines up to six different verification approaches into three sequential portfolios that are task-dependently selected with an algorithm selector. VERIABS [49] employs up to nine different verification approaches that are combined into four verification strategies and task-dependently selected by an algorithm selector. PESCO [50] ranks individual verification algorithms according to their predicted likelihood of solving a given task and then executes them sequentially in descending order. PREDATORHP [65] and UFO [66] demonstrate that parallel portfolios can also be a promising strategy when running multiple specialized algorithms at the same time. Even though previous work showed that internal combinations can be successfully applied to improve the effectiveness of a single tool, we show that similar combinations can be effectively employed to combine ‘off-the-shelf’ verifiers. This gives us the unique opportunity to further increase the number of verifiable programs by simply combining state-of-the-art verification tools.

Cooperative methods [67] distribute the workload of a single verification task among multiple algorithms to combine their strengths. For example, conditional model checking [69–72] runs two or more verifiers in sequence, while the program is reduced after every step to the state space of the program that is left unexplored by the previous algorithm. COVERTest [73, 74], a tool for test-case generation based on verification, interleaves multiple verifiers, while (partially) sharing the analysis state between algorithms. METAVAL [55] integrates verification tools for witness validation (i.e., to check whether a verifier had produced a valid result) by instrumenting the produced witness into the verified program. While cooperative methods are effective for reducing the workload of a verification task, employing cooperative methods



at the tool level requires the exchange of analysis information between tools. In general, existing verification tools are not well suited for this type of combination, which leads us to explore off-the-shelf verifier combinations. In addition, we showed that non-cooperative methods can improve the verification effectiveness without the need to adapt the employed tools.

**Combining Algorithms Beyond Software Verification.** The idea of combining algorithms to improve performance has been successfully applied in many research areas including SAT solving [75–77], constraint-satisfaction programs [78–80], and combinatorial-search problems [81]. The employed approaches traditionally focused on portfolio-based approaches [75, 76, 79], but recent techniques started to integrate algorithm selectors for either selecting single algorithms [77, 78] or portfolios of algorithms [80, 82]. For example, earlier works in SAT solving [75, 76] focused on parallel-portfolio solvers, while later works such as SATZILLA [77] further improve the solving process by selecting a task-dependent solver. However, existing techniques often employ hybrid strategies between portfolios and algorithm selection to achieve state-of-the-art performance. Therefore, Kashgarani and Kothoff [83] have recently shown that parallel portfolios are generally bottlenecked by the available resources and that a pure algorithm selector that selects a single algorithm performs better. While we observed that portfolios of software verifiers are also restricted by available resources (i.e., the performance generally stops to improve after a certain portfolio size), we found that all evaluated combination types can yield performance gains.

## 8 Conclusion

This paper describes a method to construct combinations of verification and validation tools in a systematic and modular way. The method does not require any changes to the tools that are used to construct the combinations. Given the large number of freely available verifiers and validators for C programs, there is a huge potential for improvement in effectivity and efficiency (a total of 50 verifiers and 13 validators were evaluated in SV-COMP 2024 [84]). Our experimental evaluation shows that all three considered combinations—sequential portfolio, parallel portfolio, and algorithm selection—can lead to performance improvements. The improvements can be significant although the construction does not require significant development effort, because we use COVERITEAM for the combination and execution of verification tools. Our contribution is to offer an easy way for practitioners to benefit from the available verification tools and leverage better performance from the latest research and development efforts in software verification.

**Funding** Open Access funding enabled and organized by Projekt DEAL. This work was funded in part by Deutsche Forschungsgesellschaft (DFG)—378803395 (ConVeY) and 418257054 (Coop).

**Data availability** A reproduction package including all our results is available at Zenodo [22]. Additionally, the result tables are also available on a supplementary web page for convenient browsing: <https://www.sosy-lab.org/research/coveriteam-combinations>.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.



## References

1. Beyer D, Podelski A (2022) Software model checking: 20 years and beyond. In: Principles of systems design. LNCS, vol 13660. Springer, pp 554–582. [https://doi.org/10.1007/978-3-031-22337-2\\_27](https://doi.org/10.1007/978-3-031-22337-2_27)
2. Hoare CAR (2003) The verifying compiler: a grand challenge for computing research. J. ACM 50(1):63–69. <https://doi.org/10.1145/602382.602403>
3. Clarke EM, Henzinger TA, Veith H, Bloem R (2018) Handbook of model checking. Springer, Berlin. <https://doi.org/10.1007/978-3-319-10575-8>
4. Jhala R, Majumdar R (2009) Software model checking. ACM Comput Surv. doi 10(1145/1592434):1592438
5. Beyer D (2022) Progress on software verification: SV-COMP 2022. In: Proceedings of TACAS (2), LNCS, vol 13244. Springer, pp 375–402. [https://doi.org/10.1007/978-3-030-99527-0\\_20](https://doi.org/10.1007/978-3-030-99527-0_20)
6. Beckert B, Hähnle R (2014) Reasoning and verification: state of the art and current trends. IEEE Intell Syst 29(1):20–29. <https://doi.org/10.1109/MIS.2014.3>
7. Beyer D, Gulwani S, Schmidt D (2018) Combining model checking and data-flow analysis. In: Handbook of model checking. Springer, pp 493–540. [https://doi.org/10.1007/978-3-319-10575-8\\_16](https://doi.org/10.1007/978-3-319-10575-8_16)
8. Garavel H, ter Beek MH, van de Pol J (2020) The 2020 expert survey on formal methods. In: Proceedings of FMICS. LNCS, vol 12327. Springer, pp 3–69. [https://doi.org/10.1007/978-3-030-58298-2\\_1](https://doi.org/10.1007/978-3-030-58298-2_1)
9. Ball T, Rajamani SK (2002) The SLAM project: debugging system software via static analysis. In: Proceedings of POPL. ACM, pp 1–3. <https://doi.org/10.1145/503272.503274>
10. Khoroshilov AV, Mutilin VS, Petrenko AK, Zakharov V (2009) Establishing Linux driver verification process. In: Proceedings of Ershov memorial conference. LNCS, vol 5947. Springer, pp 165–176. [https://doi.org/10.1007/978-3-642-11486-1\\_14](https://doi.org/10.1007/978-3-642-11486-1_14)
11. Chong N, Cook B, Eidelman J, Kallas K, Khazem K, Monteiro FR, Schwartz-Narbonne D, Tasiran S, Tautschnig M, Tuttle MR (2021) Code-level model checking in the software development workflow at Amazon Web Services. Softw Pract Exp 51(4):772–797. <https://doi.org/10.1002/spe.2949>
12. Calcagno C, Distefano D, Dubreil J, Gabi D, Hooimeijer P, Luca M, O’Hearn PW, Papakonstantinou I, Purbrick J, Rodriguez D (2015) Moving fast with software verification. In: Proceedings of NFM. LNCS, vol 9058. Springer, pp 3–11. [https://doi.org/10.1007/978-3-319-17524-9\\_1](https://doi.org/10.1007/978-3-319-17524-9_1)
13. Beyer D, Keremoglu ME (2011) CPACHECKER: a tool for configurable software verification. In: Proceedings of CAV. LNCS, vol 6806. Springer, pp 184–190. [https://doi.org/10.1007/978-3-642-22110-1\\_16](https://doi.org/10.1007/978-3-642-22110-1_16)
14. Dangel M, Löwe S, Wendler P (2015) CPACHECKER with support for recursive programs and floating-point arithmetic (competition contribution). In: Proceedings of TACAS. LNCS, vol 9035. Springer, pp 423–425. [https://doi.org/10.1007/978-3-662-46681-0\\_34](https://doi.org/10.1007/978-3-662-46681-0_34)
15. Gadelha MYR, Monteiro FR, Cordeiro LC, Nicole DA (2019) ESBMC v6.0: verifying C programs using k-induction and invariant inference (competition contribution). In: Proceedings of TACAS (3). LNCS, vol 11429. Springer, pp 209–213. [https://doi.org/10.1007/978-3-030-17502-3\\_15](https://doi.org/10.1007/978-3-030-17502-3_15)
16. Huberman BA, Lukose RM, Hogg T (1997) An economics approach to hard computational problems. Science 275(7):51–54. <https://doi.org/10.1126/science.275.5296.51>
17. Rice JR (1976) The algorithm selection problem. Adv Comput 15:65–118. [https://doi.org/10.1016/S0065-2458\(08\)60520-3](https://doi.org/10.1016/S0065-2458(08)60520-3)
18. Beyer D, Kanav S (2022) CoVeriTEAM: on-demand composition of cooperative verification systems. In: Proceedings of TACAS. LNCS, vol 13243. Springer, pp 561–579. [https://doi.org/10.1007/978-3-030-99524-9\\_31](https://doi.org/10.1007/978-3-030-99524-9_31)
19. Beyer D, Kanav S, Wachowitz H. Source-code repository of CoVeriTEAM. <https://gitlab.com/sosy-lab/software/coveriteam>. Accessed 09 Feb 2023
20. Beyer D, Kanav S, Wachowitz H (2023) CoVeriTEAM service: verification as a service. In: Proceedings of ICSE, companion. IEEE, pp 21–25. <https://doi.org/10.1109/ICSE-Companion58688.2023.00017>
21. Beyer D, Kanav S, Richter C (2022) Construction of verifier combinations based on off-the-shelf verifiers. In: Proceedings of FASE. Springer, pp 49–70. [https://doi.org/10.1007/978-3-030-99429-7\\_3](https://doi.org/10.1007/978-3-030-99429-7_3)
22. Beyer D, Kanav S, Kleinert T, Richter C (2023) Reproduction package for FMSD article ‘Construction of verifier combinations from off-the-shelf components’. Zenodo. <https://doi.org/10.5281/zenodo.7838348>
23. Wendler P (2013) CPACHECKER with sequential combination of explicit-state analysis and predicate analysis (competition contribution). In: Proceedings of TACAS. LNCS, vol 7795. Springer, pp 613–615. [https://doi.org/10.1007/978-3-642-36742-7\\_45](https://doi.org/10.1007/978-3-642-36742-7_45)
24. Heizmann M, Chen YF, Dietsch D, Greitschus M, Hoenicke J, Li Y, Nutz A, Musa B, Schilling C, Schindler T, Podelski A (2018) ULTIMATE AUTOMIZER and the search for perfect interpolants (competition contribution). In: Proceedings of TACAS (2). LNCS, vol 10806. Springer, pp 447–451. [https://doi.org/10.1007/978-3-319-89963-3\\_30](https://doi.org/10.1007/978-3-319-89963-3_30)

25. Kotoun M, Peringer P, Šoková V, Vojnar T (2016) Optimized PREDATORHP and the SV-COMP heap and memory safety benchmark (competition contribution). In: Proceedings of TACAS. LNCS, vol 9636. Springer, pp 942–945. [https://doi.org/10.1007/978-3-662-49674-9\\_66](https://doi.org/10.1007/978-3-662-49674-9_66)
26. Holík L, Kotoun M, Peringer P, Šoková V, Trtík M, Vojnar T (2016) Predator shape analysis tool suite. Proceedings of HVC. LNCS 10028:202–209. [https://doi.org/10.1007/978-3-319-49052-6\\_13](https://doi.org/10.1007/978-3-319-49052-6_13)
27. Richter C, Hüllermeier E, Jakobs MC, Wehrheim H (2020) Algorithm selection for software validation based on graph kernels. *Autom Softw Eng* 27(1):153–186. <https://doi.org/10.1007/s10515-020-00270-x>
28. Richter C, Wehrheim H (2020) Attend and represent: a novel view on algorithm selection for software verification. In: Proceedings of ASE, pp 1016–1028. <https://doi.org/10.1145/3324884.3416633>
29. Beyer D, Dangl M (2018) Strategy selection for software verification based on boolean features: a simple but effective approach. In: Proceedings of ISO/LA. LNCS, vol 11245. Springer, pp 144–159. [https://doi.org/10.1007/978-3-030-03421-4\\_11](https://doi.org/10.1007/978-3-030-03421-4_11)
30. Demyanova Y, Pani T, Veith H, Zuleger F (2017) Empirical software metrics for benchmarking of verification tools. *Formal Methods Syst Des* 50(2–3):289–316. <https://doi.org/10.1007/s10703-016-0264-5>
31. Beyer D, Dangl M, Dietsch D, Heizmann M, Lemberger T, Tautschnig M (2022) Verification witnesses. *ACM Trans Softw Eng Methodol* 31(4):57:1–57:69. <https://doi.org/10.1145/3477579>
32. Beyer D, Dangl M, Dietsch D, Heizmann M, Stahlbauer A (2015) Witness validation and stepwise testification across software verifiers. In: Proceedings of FSE. ACM, pp 721–733. <https://doi.org/10.1145/2786805.2786867>
33. Beyer D, Dangl M, Dietsch D, Heizmann M (2016) Correctness witnesses: exchanging verification results between verifiers. In: Proceedings of FSE. ACM, pp 326–337. <https://doi.org/10.1145/2950290.2950351>
34. Beyer D, Dangl M, Lemberger T, Tautschnig M (2018) Tests from witnesses: execution-based validation of verification results. In: Proceedings of TAP. LNCS, vol 10889. Springer, pp 3–23. [https://doi.org/10.1007/978-3-319-92994-1\\_1](https://doi.org/10.1007/978-3-319-92994-1_1)
35. Beyer D, Dangl M (2016) Verification-aided debugging: an interactive web-service for exploring error witnesses. In: Proceedings of CAV (2). LNCS, vol 9780. Springer, pp 502–509. [https://doi.org/10.1007/978-3-319-41540-6\\_28](https://doi.org/10.1007/978-3-319-41540-6_28)
36. Beyer D (2015) Software verification and verifiable witnesses (Report on SV-COMP 2015). In: Proceedings of TACAS. LNCS, vol 9035. Springer, pp 401–416. [https://doi.org/10.1007/978-3-662-46681-0\\_31](https://doi.org/10.1007/978-3-662-46681-0_31)
37. Beyer D, Löwe S, Wendler P (2019) Reliable benchmarking: requirements and solutions. *Int J Softw Tools Technol Transf* 21(1):1–29. <https://doi.org/10.1007/s10009-017-0469-y>
38. Kleinert T (2022) Developing a verifier based on parallel portfolio with COVERITEAM. Bachelor's Thesis, LMU Munich, Software Systems Lab
39. Demyanova Y, Veith H, Zuleger F (2013) On the concept of variable roles and its use in software analysis. In: Proceedings of FMCAD. IEEE, pp 226–230. <https://doi.org/10.1109/FMCAD.2013.6679414>
40. Apel S, Beyer D, Friedberger K, Raimondi F, Rhein A (2013) Domain types: abstract-domain selection based on variable usage. In: Proceedings of HVC. LNCS, vol 8244. Springer, pp 262–278. <https://doi.org/10.1007/978-3-319-03077-7>
41. Xu L, Hoos HH, Leyton-Brown K (2007) Hierarchical hardness models for SAT. In: International conference on principles and practice of constraint programming. Springer, pp 696–711. [https://doi.org/10.1007/978-3-540-74970-7\\_49](https://doi.org/10.1007/978-3-540-74970-7_49)
42. Kadioglu S, Malitsky Y, Sabharwal A, Samulowitz H, Sellmann M (2011) Algorithm selection and scheduling. In: Proceedings of CP. Springer, pp 454–469. [https://doi.org/10.1007/978-3-642-23786-7\\_35](https://doi.org/10.1007/978-3-642-23786-7_35)
43. Torgo L, Gama J (1996) Regression by classification. In: Brazilian symposium on artificial intelligence. Springer, pp 51–60. [https://doi.org/10.1007/3-540-61859-7\\_6](https://doi.org/10.1007/3-540-61859-7_6)
44. Chalupa M, Řečtáčková A, Mihalkovič V, Zaoral L, Strejček J (2022) SYMBIOTIC 9: string analysis and backward symbolic execution with loop folding (competition contribution). In: Proceedings of TACAS (2). LNCS, vol 13244. Springer, pp 462–467. [https://doi.org/10.1007/978-3-030-99527-0\\_32](https://doi.org/10.1007/978-3-030-99527-0_32)
45. Kröning D, Tautschnig M (2014) CBMC: C bounded model checker (competition contribution). In: Proceedings of TACAS. LNCS, vol 8413. Springer, pp 389–391. [https://doi.org/10.1007/978-3-642-54862-8\\_26](https://doi.org/10.1007/978-3-642-54862-8_26)
46. Chaudhary E, Joshi S (2019) PINAKA: symbolic execution meets incremental solving (competition contribution). In: Proceedings of TACAS (3). LNCS, vol 11429. Springer, pp 234–238. [https://doi.org/10.1007/978-3-030-17502-3\\_20](https://doi.org/10.1007/978-3-030-17502-3_20)
47. Dietsch D, Heizmann M, Nutz A, Schätzle C, Schüssele F (2020) ULTIMATE TAIPAN with symbolic interpretation and fluid abstractions (competition contribution). In: Proceedings of TACAS (2). LNCS, vol 12079. Springer, pp 418–422. [https://doi.org/10.1007/978-3-030-45237-7\\_32](https://doi.org/10.1007/978-3-030-45237-7_32)
48. Malík V, Schrammel P, Vojnar T (2020) 2LS: Heap analysis and memory safety (competition contribution). In: Proceedings of TACAS (2). LNCS, vol 12079. Springer, pp 368–372. [https://doi.org/10.1007/978-3-030-45237-7\\_22](https://doi.org/10.1007/978-3-030-45237-7_22)

49. Darke P, Agrawal S, Venkatesh R (2021) VERIABS: a tool for scalable verification by abstraction (competition contribution). In: Proceedings of TACAS (2). LNCS, vol 12652. Springer, pp 458–462. [https://doi.org/10.1007/978-3-030-72013-1\\_32](https://doi.org/10.1007/978-3-030-72013-1_32)
50. Richter C, Wehrheim H (2019) PESCO: predicting sequential combinations of verifiers (competition contribution). In: Proceedings of TACAS (3). LNCS, vol 11429. Springer, pp 229–233. [https://doi.org/10.1007/978-3-030-17502-3\\_19](https://doi.org/10.1007/978-3-030-17502-3_19)
51. Leeson W, Dwyer, M (2022) GRAVES-CPA: a graph-attention verifier selector (competition contribution). In: Proceedings of TACAS (2). LNCS, vol 13244. Springer, pp 440–445. [https://doi.org/10.1007/978-3-030-99527-0\\_28](https://doi.org/10.1007/978-3-030-99527-0_28)
52. Beyer D, Strejček J (2022) Case study on verification-witness validators: where we are and where we go. In: Proceedings of SAS. LNCS, vol 13790. Springer, pp 160–174. [https://doi.org/10.1007/978-3-031-22308-2\\_8](https://doi.org/10.1007/978-3-031-22308-2_8)
53. Ayaziová P, Chalupa M, Strejček J (2022) SYMBIOTIC- WITCH: a Klee-based violation witness checker (competition contribution). In: Proceedings of TACAS (2). LNCS, vol 13244. Springer, pp 468–473. [https://doi.org/10.1007/978-3-030-99527-0\\_33](https://doi.org/10.1007/978-3-030-99527-0_33)
54. Švejda J, Berger P, Katoen JP (2020) Interpretation-based violation witness validation for C: NitWit. In: Proceedings of TACAS. LNCS, vol 12078. Springer, pp 40–57. [https://doi.org/10.1007/978-3-030-45190-5\\_3](https://doi.org/10.1007/978-3-030-45190-5_3)
55. Beyer D, Spiessl M (2020) METAVAL: witness validation via verification. In: Proceedings of CAV. LNCS, vol 12225. Springer, pp 165–177. [https://doi.org/10.1007/978-3-030-53291-8\\_10](https://doi.org/10.1007/978-3-030-53291-8_10)
56. Collection of verification tasks. <https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks>. Accessed 01 Apr 2023
57. Beyer D (2022) SV-benchmarks: benchmark set for software verification and testing (SV-COMP 2022 and Test-Comp 2022). Zenodo. <https://doi.org/10.5281/zenodo.5831003>
58. Beyer D (2013) Second competition on software verification (Summary of SV-COMP 2013). In: Proceedings of TACAS. LNCS, vol 7795. Springer, pp 594–609. [https://doi.org/10.1007/978-3-642-36742-7\\_43](https://doi.org/10.1007/978-3-642-36742-7_43)
59. Beyer D (2012) Competition on software verification (SV-COMP). In: Proceedings of TACAS. LNCS, vol 7214. Springer, pp 504–524. [https://doi.org/10.1007/978-3-642-28756-5\\_38](https://doi.org/10.1007/978-3-642-28756-5_38)
60. Leino KRM (2010) Dafny: an automatic program verifier for functional correctness. In: Proceedings of LPAR. LNCS, vol 6355. Springer, pp 348–370. [https://doi.org/10.1007/978-3-642-17511-4\\_20](https://doi.org/10.1007/978-3-642-17511-4_20)
61. Cuoq P, Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B (2012) Frama-C. In: Proceedings of SEFM. Springer, pp 233–247. [https://doi.org/10.1007/978-3-642-33826-7\\_16](https://doi.org/10.1007/978-3-642-33826-7_16)
62. Ahrendt W, Baar T, Beckert B, Bubel R, Giese M, Hähnle R, Menzel W, Mostowski W, Roth A, Schlager S, Schmitt PH (2005) The key tool. *Softw Syst Model* 4(1):32–54. <https://doi.org/10.1007/s10270-004-0058-x>
63. Blom S, Huisman M (2014) The VerCors tool for verification of concurrent programs. In: Proceedings of FM. LNCS, vol 8442. Springer, pp 127–131. [https://doi.org/10.1007/978-3-319-06410-9\\_9](https://doi.org/10.1007/978-3-319-06410-9_9)
64. Jacobs B, Smans J, Philippaerts P, Vogels F, Penninckx W, Piessens F (2011) VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Proceedings of NFM. LNCS, vol 6617. Springer, pp 41–55. [https://doi.org/10.1007/978-3-642-20398-5\\_4](https://doi.org/10.1007/978-3-642-20398-5_4)
65. Peringer P, Šoková V, Vojnar T (2020) PREDATORHP revamped (not only) for interval-sized memory regions and memory reallocation (competition contribution). In: Proceedings of TACAS (2). LNCS, vol 12079. Springer, pp 408–412. [https://doi.org/10.1007/978-3-030-45237-7\\_30](https://doi.org/10.1007/978-3-030-45237-7_30)
66. Albarghouthi A, Li Y, Gurfinkel A, Chechik M (2012) UFO: a framework for abstraction- and interpolation-based software verification. In: Proceedings of CAV. LNCS, vol 7358. Springer, pp 672–678. [https://doi.org/10.1007/978-3-642-31424-7\\_48](https://doi.org/10.1007/978-3-642-31424-7_48)
67. Beyer D, Wehrheim H (2020) Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proceedings of ISoLA (1). LNCS, vol 12476. Springer, pp 143–167. [https://doi.org/10.1007/978-3-030-61362-4\\_8](https://doi.org/10.1007/978-3-030-61362-4_8)
68. Filliâtre JC, Paskevich A (2013) Why3: where programs meet provers. In: Programming languages and systems. Springer, pp 125–128. [https://doi.org/10.1007/978-3-642-37036-6\\_8](https://doi.org/10.1007/978-3-642-37036-6_8)
69. Beyer D, Henzinger TA, Keremoglu ME, Wendler P (2012) Conditional model checking: a technique to pass information between verifiers. In: Proceedings of FSE. ACM. <https://doi.org/10.1145/2393596.2393664>
70. Beyer D, Jakobs MC, Lemberger T, Wehrheim H (2018) Reducer-based construction of conditional verifiers. In: Proceedings of ICSE. ACM, pp 1182–1193. <https://doi.org/10.1145/3180155.3180259>
71. Beyer D, Jakobs MC (2020) Fred: conditional model checking via reducers and folders. In: Proceedings of SEFM. LNCS, vol 12310. Springer, pp 113–132. [https://doi.org/10.1007/978-3-030-58768-0\\_7](https://doi.org/10.1007/978-3-030-58768-0_7)
72. Beyer D, Jakobs MC, Lemberger T (2020) Difference verification with conditions. In: Proceedings of SEFM. LNCS, vol 12310. Springer, pp 133–154. [https://doi.org/10.1007/978-3-030-58768-0\\_8](https://doi.org/10.1007/978-3-030-58768-0_8)

73. Beyer D, Jakobs MC (2019) CoVeriTest: cooperative verifier-based testing. In: Proceedings of FASE. LNCS, vol 11424. Springer, pp 389–408. [https://doi.org/10.1007/978-3-030-16722-6\\_23](https://doi.org/10.1007/978-3-030-16722-6_23)
74. Beyer D, Jakobs MC (2021) Cooperative verifier-based testing with CoVeriTest. Int J Softw Tools Technol Transf 23(3):313–333. <https://doi.org/10.1007/s10009-020-00587-8>
75. Wotzlaw A, van der Grinten A, Speckenmeyer E, Porschen S (2012) pfolioUZK: solver description. In: Proceedings of SAT challenge p 45. <https://helda.helsinki.fi/handle/10138/34218>
76. Roussel O (2011) Description of portfolio. In: Proceedings of SAT challenge, p 46. <https://helda.helsinki.fi/handle/10138/34218>
77. Xu L, Hutter F, Hoos HH, Leyton-Brown K (2008) SATzilla: Portfolio-based algorithm selection for SAT. JAIR 32:565–606. <https://doi.org/10.1613/jair.2490>
78. Minton S (1996) Automatically configuring constraint satisfaction programs: a case study. Constraints 1(1–2):7–43. <https://doi.org/10.1007/BF00143877>
79. Bordeaux L, Hamadi Y, Samulowitz H (2009) Experiments with massively parallel constraint solving. In: Proceedings of IJCAI. <https://www.ijcai.org/Proceedings/09/Papers/081.pdf>
80. Yun X, Epstein SL (2012) Learning algorithm portfolios for parallel execution. In: Proceedings of LION. Springer, pp 323–338. [https://doi.org/10.1007/978-3-642-34413-8\\_23](https://doi.org/10.1007/978-3-642-34413-8_23)
81. Kotthoff L (2016) Algorithm selection for combinatorial search problems: a survey. In: Data mining and constraint programming—foundations of a cross-disciplinary approach. LNCS, vol 10101. Springer, pp 149–190. [https://doi.org/10.1007/978-3-319-50137-6\\_7](https://doi.org/10.1007/978-3-319-50137-6_7)
82. Lindauer M, Hoos H, Hutter F (2015) From sequential algorithm selection to parallel portfolio selection. In: Proceedings of LION. Springer, pp 1–16. [https://doi.org/10.1007/978-3-319-19084-6\\_1](https://doi.org/10.1007/978-3-319-19084-6_1)
83. Kashgarani H, Kotthoff L (2021) Is algorithm selection worth it? Comparing selecting single algorithms and parallel execution. In: AAAI workshop on meta-learning and MetaDL challenge. PMLR, pp 58–64. <https://proceedings.mlr.press/v140/kashgarani21a.html>
84. Beyer D (2024) State of the art in software verification and witness validation: SV-COMP 2024. In: Proceedings of TACAS (3). LNCS, vol 14572. Springer, pp 299–329. [https://doi.org/10.1007/978-3-031-57256-2\\_15](https://doi.org/10.1007/978-3-031-57256-2_15)

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



# CoVeriTeam Service: Verification as a Service

Dirk Beyer <sup>LMU</sup>  
LMU Munich, Germany

Sudeep Kanav <sup>LMU</sup>  
LMU Munich, Germany

Henrik Wachowitz <sup>LMU</sup>  
LMU Munich, Germany

**Abstract**—The research community has developed numerous tools for solving verification problems, but we are missing a common web interface for executing them. This means, users have to commit to install and execute each new tool (version) on their local machine. We propose to use CoVeriTeam Service to make it easy for verification researchers to experiment with new verification tools. CoVeriTeam has already unified the command-line interface, and reduced the burden by taking care of tool installation and isolated execution. The new web service in addition enables tool developers to make their tools accessible on the web and users to include verification tools in their work flow. There are already further applications of our service: The 2023 competitions on software verification and testing used the service for their integration testing, and we propose to use CoVeriTeam Service for incremental verification as part of a continuous-integration process.

Demonstration video: <https://youtu.be/0Ao0ZogSu1U>

Demonstration service: <https://coveriteam-service.sosy-lab.org>

**Index Terms**—Cooperative Verification, Tool Development, Incremental Verification, Software Verification, Automatic Verification, Verification Tools, Web Service, API, Continuous Integration

## I. INTRODUCTION

Numerous automated verification and testing tools have been developed in the last few decades. This is attested by the growing number of participants in the competitions on software verification (SV-COMP) [5] and testing (Test-Comp) [4].

Consider a verification researcher or student who wants to experiment with a verification tool. On the one hand, a larger number of tools provides more opportunities to such a user. On the other hand, it requires considerable effort to figure out how to execute each tool and interpret the results. Additionally, there might be a mismatch between the configuration of the user's system and the required configuration to execute the tool. Or, the user might not want to execute an untrusted verification tool locally due to security concerns. Even with numerous choices available for the verification tools, it is not straightforward to use them.

There are arguments regarding the developers of verification tools pointing in the same direction. A lot of effort goes in developing the tools; but even after spending so much effort in developing excellent tools, the tools are not easily accessible.

This work aims at improving the accessibility of automated verification tools. We propose a web service that enables remote execution of verification tools. The solution is based on CoVeriTeam [7], which provides already a common command-line interface to verification tools. A common web interface makes it easier for users to experiment with arbitrary verification tools, as well as for tool developers to benefit from making their tools more accessible to new users. The web service

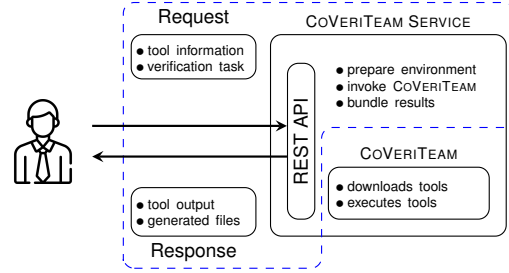


Fig. 1: Abstract view of the CoVeriTeam Service

liberates users from the concerns of configuring the local system. Additionally, running tools remotely eliminates security concerns arising from local execution of untrusted code.

CoVeriTeam [7] provides a common interface to verification tools through the means of verification actors and artifacts. It manages download, execution, and processing input and output of these tools. It provides a library of actors for many publicly available verification and testing tools. Also, it provides means to integrate a new tool easily via BENCHEXEC [16].

Developing a web service for CoVeriTeam allows us to reuse the infrastructure that was developed for CoVeriTeam. If users want to execute a verification tool remotely, they can simply invoke CoVeriTeam with the required inputs and the option for remote execution, and CoVeriTeam manages the rest. Alternatively, the service can be called directly via its REST API, making it suitable for integration into other applications. Figure 1 shows the abstract view of our solution. A user sends the information about the tool and the verification task to the service; the service prepares the environment, invokes CoVeriTeam, and bundles the results of the execution; and CoVeriTeam manages tool download and its execution.

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

- 1) *Remote execution*: a web service to remotely execute publicly available verification tools and their combinations,
- 2) *Incremental verifier*: a web service for incremental verification using CoVeriTeam Service as a micro-service,
- 3) *Easy access to verification tools*: a solution allowing tool developers to provide easy access to their verification tools,
- 4) *Reuse*: an open-source Python implementation and a reproduction package [11, 12].

**Impact.** This service was part of the continuous-integration pipeline of the competitions SV-COMP and Test-Comp 2023. The service was used to make sure that a (new version of a)



tool is only integrated if it can be successfully executed in the specified competition environment. This makes it possible to find issues with the participating tool archive early on, thus saving the effort required by the competition organizer and participants to locate and solve the issues by manual inspection.

**Related Work.** *Electronic Tools Integration* (ETI) [23, 25] was conceived as a platform to allow users to access tools through the internet. It was intended to serve as a site for testing, presenting, evaluation, and benchmarking tools.

*Model Checking as a Service* (MCaaS) [21] aims to hide the model checking entirely from engineers. The authors aim to provide a ready made solution for a specific engineering workflow with limited configuration.

The (deprecated) RiSE4Fun [3] service is a web front-end to web services on formal methods. CoVeriTEAM Service is not a web front-end, but a backend service. It could have been integrated as one of the backend services in RiSE4Fun, acting as a generic interface to many verification tools, similar to our own web UI.

Unite [26] aims to provide easier access to analysis tools by converting them to web services. It uses *open services for lifecycle collaboration* (OSLC) to create web services from software-analysis tools. While ETI, MCaaS, and our service provide a remote interface to analysis tools, the focus of Unite is to provide a framework to setup a web service for *one tool at a time*. This requires manual setup and configuration of the tool and its arguments. Once configured, the OSLC endpoint integrates with respective OSLC clients that are able to mirror the tool's interface.

Our approach decouples (a) integrating a new tool from (b) providing a web service: (a) CoVeriTEAM integrates the analysis tools and provides the interface to the tools; (b) CoVeriTEAM SERVICE provides a web-service interface using a REST API. The interface to our web service is uniform, regardless of the tool a user wants to execute.

Many verification tools have implemented some form of web front-end [6, 18, 20, 22], stating a clear demand for web-accessibility. With CoVeriTEAM SERVICE, we provide one uniform interface for all tools. We present a web UI in Sect. IV, where users can already select from 21 verification tools and run them remotely.

**Running Example.** Users want to verify a program. They want to use a verification tool to achieve it; but they do not want to execute the tool locally. There could be multiple reasons for the aversion to the local execution of the tools, e.g., ease of use: the user does not want to install the tool and its dependencies, or the tool might require different packages than available on the local machine, or security concerns: the user does not want to run an untrusted tool on the local machine. Our running example considers the case that our user wants to execute the verification tool CPACHECKER remotely.

## II. BACKGROUND: CoVeriTEAM

CoVeriTEAM SERVICE uses CoVeriTEAM as execution backend. CoVeriTEAM [7, 10] provides a common interface to verification

```
// Create verifier from an actor definition
verifier = ActorFactory.create(ProgramVerifier,
    "cpachecker.yml", "2.1");
// Prepare inputs
prog = ArtifactFactory.create(Program,
    "test02.c", ILP32);
spec =
    ArtifactFactory.create(BehaviorSpecification,
        "unreach-call.prp");
inputs = {"program":prog, "spec":spec};

// Execute the verifier on the inputs
result = execute(verifier, inputs);
```

Listing 1: A CoVeriTEAM program to execute a verifier

```
actor_name: "cpachecker"
toolinfo_module: "cpachecker.py"
archives:
  - version: "2.1"
    doi: "10.5281/zenodo.5720557"
    options: ["-svcomp22", "-timelimit", "900_s"]
```

Listing 2: An example verification actor in YAML format

tools. It considers verification tools as *verification actors*, and the input and output of these tools as *verification artifacts*. The behavior of CoVeriTEAM is defined by a CoVeriTEAM program, written in a simple domain-specific language for the construction and execution of tool combinations.

Listing 1 shows an example CoVeriTEAM program (.cvt file), which constructs first an actor of type *ProgramVerifier* from an external-actor definition that is provided in file *cpachecker.yml*. Then it creates two artifacts of types *Program* and *BehaviorSpecification* that are prepared as input mapping for the execution of the verifier on these inputs. CoVeriTEAM downloads the tool, assembles the command for the tool execution, executes the tool (in a controlled and isolated BENCHEXEC [16] container), and processes the output generated by the tool to extract the output artifacts.

Listing 2 shows the external-actor definition (.yml file) that was used in the above CoVeriTEAM program. (Internal actors are not considered here: they are the combinations constructed in the CoVeriTEAM program.) An external-actor definition specifies the name of the actor, the BENCHEXEC [16] tool-info module that is used to assemble the tool command line and parse the output produced by the tool, the version, the tool archive (by DOI or URL), command-line options, and resource limits to enforce during the execution.

External-actor definitions can be defined by the developers of the external verification tools and serve as the interface to their verification tools (see CoVeriTEAM's [library of external actors](#)).

## III. APPROACH: CoVeriTEAM SERVICE

CoVeriTEAM SERVICE and the service for incremental verification are based on a REST API. The client communicates with the services using HTTP.

**CoVeriTEAM SERVICE.** There are three ways to execute a job using CoVeriTEAM SERVICE: (1) via CoVeriTEAM, by using the same command line, just with an additional option `--remote` appended, (2) via HTTP, by writing a job specification in a JSON file (Listing 3) and send a POST request to CoVeriTEAM

```
{
  "cvt_program": "verifier.cvt",
  "coveriteam_inputs": {
    "verifier_path": "cpachecker.yml",
    "program_path": "test02.c",
    "specification_path": "unreach-call.prp",
  },
  "working_directory": "coveriteam/examples",
}
```

Listing 3: Example JSON showing the input for the service

```
curl \
  --form "args=<listing3.json" \
  --form cpachecker.yml=@cpachecker.yml \
  --form test02.c=@test02.c \
  --form verifier.cvt=@verifier.cvt \
  --form unreach-call.prp=@unreach-call.prp \
  --output cvt_remote_output.zip \
  https://coveriteam-service.sosy-lab.org/execute
```

Listing 4: Example CURL request for the service

SERVICE, e.g., via CURL (Listing 4), or (3) via the web UI, by visiting the service using a web browser and completing the form.

COVERITEAM SERVICE receives the input via one of the three ways, performs consistency checks on the input, assembles the command for COVERITEAM, executes the given COVERITEAM program on the server, and sends back the output artifacts.

**Service for Incremental Verification.** While tests and lightweight linters are regularly used in continuous integration (CI), verification tools are more difficult to integrate, due to their resource requirements. Our service makes it possible to delegate the verification load from the CI machine to a dedicated machine for the verification service via COVERITEAM SERVICE. Furthermore, incremental verification (IV) aims at speeding up the verification of the current program version by reusing knowledge from verifying a previous version [15, 24].

We offer a solution that combines the two approaches: a *service* for *incremental* verification. The solution is based on the following components: (1) a verifier that can export and import the knowledge learned during a verification run, (2) a store to save and retrieve this information, and (3) a *manager* to connect the above two. In our solution, COVERITEAM SERVICE is used as a micro-service to execute the verifier; and the service for IV manages the store and interacts with COVERITEAM SERVICE.

Figure 2 illustrates the data flow of our solution. The service for incremental verification stores and retrieves the knowledge

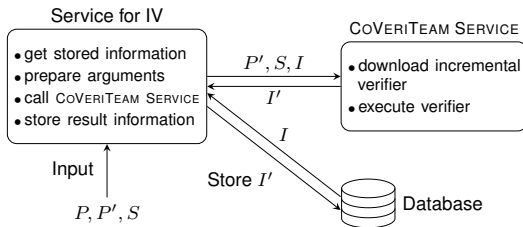


Fig. 2: Architecture of the service for incremental verification

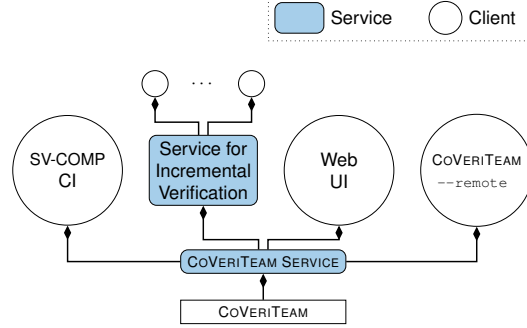


Fig. 3: Various clients of the COVERITEAM service

exported by the verification tool so that it can be reused later to assist in verification. The service takes as input a current program  $P'$ , its previous version  $P$ , and the specification  $S$ . It first checks if some reusable verification information  $I$  for  $P$  and  $S$  is stored in the database. If available, it retrieves the stored information and calls COVERITEAM SERVICE passing it the current program  $P'$ , the specification  $S$ , and the verification information  $I$ . If the information is absent, then the task is treated as a fresh verification task. COVERITEAM SERVICE then executes the verifier, and returns the result. Before returning this result to the user, the service for incremental verification extracts and stores relevant information obtained from the result.

We explain an instantiation of this construction in Sect. IV using CPACHECKER as the verifier. It is straightforward to adapt this to another verifier that supports export and import of any information reusable for verification.

#### IV. APPLICATIONS AND USE CASES

A detailed evaluation of COVERITEAM through case studies and experiments has been presented in our previous work [7, 8]. Here we present some applications and use cases of the service.

**Remote Execution of Verification Tools.** Using our publicly available installation of a COVERITEAM SERVICE instance at <https://coveriteam-service.sosy-lab.org>, a user can execute verification tools and their combinations remotely without the need to install and execute them locally. The service can be called using (1) the client integrated in COVERITEAM, by simply adding the option `--remote` to a command (this option instructs COVERITEAM to execute *remotely* via the service, whereas without this option execution happens *locally*), (2) a CURL command (Listing 4), or (3) a web page allowing users to execute tools that are included in COVERITEAM's library of external actors. Figure 3 shows the currently implemented clients of the service.

This service has been hit more than 930 times within a month of going online. Many of these hits were from the participants of SV-COMP and Test-Comp who wanted to test their tools on a machine similar to the ones used in the competition.

**Continuous Integration for Competitions.** The service has been used in SV-COMP and Test-Comp this year as part of the continuous-integration (CI) pipeline of the competition

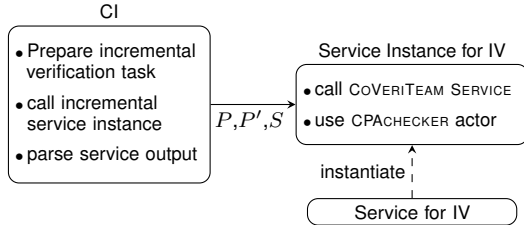


Fig. 4: Workflow of CI for incremental verification

repositories. Submitting a zipped tool archive triggers the CI pipeline, which calls CoVeriTEAM SERVICE to execute the tool on test programs. The feedback from the CI enables the participants to fix issues with their tool archives without asking the competition organizer for test runs. This CI has helped reducing the workload of the competition organizer and participants for debugging issues with tool executions.

Additionally, tool developers can integrate a call to this service in their own CI. This ensures that the tool archive of the latest version is readily executable on a different machine (e.g., a competition environment), thus reducing the effort required for packaging and testing the tool for reproducibility.

**CI for Incremental Verification.** Software is developed incrementally. It is a common practice to integrate tests and linters in CI scripts that are triggered either on each commit to the source code or in regular intervals, to control software quality.

Formal verification is a candidate for inclusion in the CI as well, but generally it is too resource-intensive to be executed frequently. Moreover, many open-source projects execute their CI using free service plans, which provide limited computational resources. CoVeriTEAM SERVICE and our service for incremental verification can address this problem by delegating resource-intensive computations to a separate service.

To evaluate our approach, we used CPACHECKER as a backend verifier in the service for incremental verification. Figure 4 illustrates the workflow of this use case. A user commits some changes to program  $P$ , yielding the modified program  $P'$ . The CI script first prepares the incremental verification task, i.e., assembles a request containing  $P$ ,  $P'$ , and the specification  $S$ , and secondly calls the service for incremental verification to solve this incremental-verification task.

#### V. INTEGRATION OF TOOLS

**Easy Integration of Verification Tools.** To integrate a new verification tool into CoVeriTEAM SERVICE, one only needs to integrate it into CoVeriTEAM, i.e., create a tool-info module and an external-actor definition. Many tools are already integrated because they are readily available in CoVeriTEAM [7].

**Construct Service for Verification Tools.** We envision CoVeriTEAM SERVICE to be instantiated by tool developers to provide easy access to their own tools, on their own server, independently from our instance of CoVeriTEAM SERVICE. To achieve this, one needs to create a container image or a virtual

machine with the required operating system and packages, and host CoVeriTEAM SERVICE in it. This would allow an arbitrary user to connect to their new instance of CoVeriTEAM SERVICE to execute the verification tool under consideration. We have tried to make hosting the service straightforward, and provide instructions and the required configuration files (see Sect. VI).

**Limitations.** Our service is based on CoVeriTEAM [7] and BENCHEXEC [16], which require a GNU/Linux-based system providing access to cgroups. The tools from SV-COMP and Test-Comp are only a few examples of the supported tools.

#### VI. HOSTING AN INSTANCE

We provide three different options to create and host an instance of CoVeriTEAM SERVICE:

- 1) **Launching a virtual machine (VM):** The repository includes a `Vagrantfile`, which can be used by Vagrant [19] to create a fresh VM with the service installed. Additionally, we provide a ready-to-use VM as artifact.
- 2) **Launching a container:** We provide a `Containerfile` and launch scripts in the repository. Using a container manager like PODMAN [2] (or DOCKER [1]), an image containing CoVeriTEAM SERVICE can quickly be created and started.
- 3) **Integrating with an existing web server:** The service can be integrated with WSGI-compatible web servers [17]. The repository contains an example integrating the service with an Apache web server.

#### VII. CONCLUSION

Many excellent verification tools are publicly available, but installation requirements as well as resource limitations hinder the integration of verification tools into development processes. We presented CoVeriTEAM SERVICE, a web service for software verification. Our solution aims to ease the effort required by a user to start working with a verification tool, supports continuous integration by enabling delegation to a service, and provides a mechanism to tool developers to make their tools easily accessible. The service has already found a user group: 930 hits within the first month of service. The continuous-integration pipeline of the competitions SV-COMP and Test-Comp is based on our new web service.

#### DECLARATIONS

**Data-Availability Statement.** We used CoVeriTEAM [10, 13] version 1.0 and CoVeriTEAM SERVICE [11, 14] version 1.1, which are both open source and released under the Apache 2 license. We also host an instance of the service publicly available for experimentation: <https://coveriteam-service.sosy-lab.org>. A demo of the service is available on YouTube [9]. A ready to use reproduction package containing a virtual machine hosting the service is available at Zenodo [12].

**Funding Statement.** This work was funded by the Deutsche Forschungsgesellschaft (DFG) — 378803395 (ConVeY).

**Acknowledgement.** We thank Nian-Ze Lee for the valuable feedback on this article, Klara Cimbalnik for the implementation of the Web UI, and the SV-COMP community for their valuable feedback on experimenting with CoVeriTEAM SERVICE.



## REFERENCES

- [1] Docker. <https://www.docker.com/>, accessed: 2023-02-09
- [2] Podman. <https://github.com/containers/podman>, accessed: 2023-02-09
- [3] Ball, T., de Halleux, P., Swamy, N., Leijen, D.: Increasing human-tool interaction via the web. In: Proc. PASTE. pp. 49–52. ACM (2013). doi:10.1145/2462029.2462031
- [4] Beyer, D.: Advances in automatic software testing: Test-Comp 2022. In: Proc. FASE. pp. 321–335. LNCS 13241, Springer (2022). doi:10.1007/978-3-030-99429-7\_18
- [5] Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). pp. 375–402. LNCS 13244, Springer (2022). doi:10.1007/978-3-030-99527-0\_20
- [6] Beyer, D., Dresler, G., Wendler, P.: Software verification in the Google App-Engine cloud. In: Proc. CAV. pp. 327–333. LNCS 8559, Springer (2014). doi:10.1007/978-3-319-08867-9\_21
- [7] Beyer, D., Kanav, S.: CoVeriTEAM: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. LNCS 13243, Springer (2022). doi:10.1007/978-3-030-99524-9\_31
- [8] Beyer, D., Kanav, S., Richter, C.: Construction of verifier combinations based on off-the-shelf verifiers. In: Proc. FASE. pp. 49–70. Springer (2022). doi:10.1007/978-3-030-99429-7\_3
- [9] Beyer, D., Kanav, S., Wachowitz, H.: Demonstration video of CoVeriTEAM SERVICE. <https://youtu.be/0Ao0ZogSu1U>, accessed: 2023-02-16
- [10] Beyer, D., Kanav, S., Wachowitz, H.: Source-code repository of CoVeriTEAM. <https://gitlab.com/sosy-lab/software/coveriteam>, accessed: 2023-02-09
- [11] Beyer, D., Kanav, S., Wachowitz, H.: Source-code repository of CoVeriTEAM SERVICE. <https://gitlab.com/sosy-lab/software/coveriteam-service>, accessed: 2023-02-09
- [12] Beyer, D., Kanav, S., Wachowitz, H.: Reproduction package for the ICSE 2023 article ‘CoVeriTEAM service: Verification as a service’. Zenodo (2023). doi:10.5281/zenodo.7635848
- [13] Beyer, D., Kanav, S., Wachowitz, H.: CoVeriTEAM release 1.0. Zenodo (2023). doi:10.5281/zenodo.7635975
- [14] Beyer, D., Kanav, S., Wachowitz, H.: CoVeriTEAM Service release 1.1. Zenodo (2023). doi:10.5281/zenodo.7635969
- [15] Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proc. FSE. pp. 389–399. ACM (2013). doi:10.1145/2491411.2491429
- [16] Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). doi:10.1007/s10009-017-0469-y
- [17] Eby, P.J.: PEP 3333 – Python web-server gateway interface v1.0.1. Tech. rep., <https://peps.python.org/pep-3333/>, accessed: 2023-02-09
- [18] Esen, Z., Rümmer, P.: TriCERA: Verifying C programs using the theory of heaps. In: Proc. FMCAD. pp. 360–391. TU Wien Academic Press (2022). doi:10.34727/2022/isbn.978-3-85448-053-2
- [19] Hashimoto, M.: Vagrant. <https://github.com/hashicorp/vagrant>, accessed: 2023-02-09
- [20] Heizmann, M., Christ, J., Dietsch, D., Ermis, E., Hoenicke, J., Lindemann, M., Nutz, A., Schilling, C., Podelski, A.: ULTIMATE AUTOMIZER with SMTInterpol (competition contribution). In: Proc. TACAS. pp. 641–643. LNCS 7795, Springer (2013). doi:10.1007/978-3-642-36742-7\_53
- [21] Horváth, B., Graics, B., Hajdu, Á., Micskei, Z., Molnár, V., Ráth, I., Andolfato, L., v. Gomes, I., Karban, R.: Model checking as a service: Towards pragmatic hidden formal methods. In: Proc. MODELS Companion. pp. 37:1–37:5. ACM (2020). doi:10.1145/3417990.3421407
- [22] Macedo, N., Cunha, A., Pereira, J., Carvalho, R., Silva, R., Paiva, A.C.R., Ramalho, M.S., Silva, D.: Experiences on teaching ALLOY with an automated assessment platform. Science of Computer Programming **211**, 102690 (2021). doi:10.1016/j.scico.2021.102690
- [23] Margaria, T., Nagel, R., Steffen, B.: Remote integration and coordination of verification tools in JETI. In: Proc. ECBS. pp. 431–436 (2005). doi:10.1109/ECBS.2005.59
- [24] Rothenberg, B., Dietsch, D., Heizmann, M.: Incremental verification using trace abstraction. In: Proc. SAS. pp. 364–382. LNCS 11002, Springer (2018). doi:10.1007/978-3-319-99725-4\_22
- [25] Steffen, B., Margaria, T., Braun, V.: The Electronic Tool Integration platform: Concepts and design. STTT **1**(1-2), 9–30 (1997). doi:10.1007/s100090050003
- [26] Vašíček, O., Fiedor, J., Kratochvíla, T., Bohuslav, K., Smrčka, A., Vojnar, T.: Unite: An Adapter for Transforming Analysis Tools to Web Services via OSLC. In: Proc. ESEC/FSE. ACM (2022). doi:10.1145/3540250.3558939